Learning geometric phase field representations

Yannick Kees

Born 14.05.1998 in Daun, Germany

18th October 2022

Master's Thesis Mathematics

Advisor: Prof. Dr. Martin Rumpf

Second Advisor: Prof. Dr. Tim Laux

Institut für Numerische Simulation

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Acknowledgments

With this page, I would like to thank all the people who contributed in different ways to the success of this work.

First and foremost, I would like to thank my advisor Prof. Dr. Martin Rumpf, for introducing me to this topic and for his constant support throughout the last year.

I want to thank Prof. Dr. Tim Laux for taking the role of my second advisor.

I would like to thank Josua Sassen for taking the time to answer all my questions on this topic and for the many enlightening discussions during this thesis.

I am very grateful to Jonas Arruda for his many helpful remarks regarding my thesis.

I also want to thank the entire research group of Prof. Rumpf, who have been very welcoming and supportive.

Furthermore, I would like to thank my former math teacher Maria Gail, without whom I would never have studied mathematics in the first place.

Finally, I would like to thank my family, who always supported me in all my decisions.

Contents

Introduction

The area of computer graphics is a field of mathematics and computer science that deals with creating and manipulating visual content. One central question in computer graphics is the best way to represent three-dimensional data effectively. Most conventional approaches approximate the object's surface discretely, for example, with meshes. One problem with these approaches is that spatial discretization limits them, just as the display of an image is limited by its number of pixels. Another popular approach is implicit representation, in which the object is written as the level set of an appropriate function. Instead of discretizing the output domain, we can now discretize the space of available functions. In this thesis, we deal with how to find such a function. To this end, we place a particular focus on **phase field functions**. These functions are, for the most part, constant, with a smooth transition along the surface where the value of the function changes. We distinguish between two different models: the Modica-Mortola approach, in which there are two different phases for an interior and exterior, and the Ambrosio-Tortorelli approach, in which there is only one phase.
Calculating these functions is very challenging. Therefore we use deep learning, i.e., neural networks, to approximate the phase fields. The starting points for all our calculations are sets of points sampled from the surfaces of the objects. Learning implicit functions using the Modica-Mortola approach has been introduced in [Lip21]. We will derive the Ambrosio-Tortorelli approach in this work based on [AT90]. Using this new approach, we will also be able to process open surfaces, which was impossible before.
In order to be able to use neural networks for any task, there are three points that we need to consider. The first one is how we design the architecture of a neural network. This part has already been studied in depth, and we will build on these results.
The second part is to consider the available ground truth/training data. Based on that, we can go to the third point, choosing a loss functional that the network should aim to minimize.
To do this, we distinguish two different tasks. The first goal will be to train a network to match the phase field for a single 3D object. In the second step, we will train a network that can represent the phase fields for several objects at once. Therefore, the network receives additional object-specific input. For most of the current work, the ground truth data in both cases is already an implicit function that is approximated by a neural network function. In this case, the loss functional can be a metric between the given function and the learned function. However, if the available data consists of only a set of points sampled from the object's surface, this becomes more difficult. Therefore, we will derive a new method for the case in which several phase field representations are learned at once based on autoencoders.
In this thesis, our primary goal will be finding suitable loss functions for all these tasks. We will

then test these losses and their different parameters on objects in 2D and 3D and show how to construct simple datasets for this task.
Since the network functions we consider are all completely visualizable, this work also provides insights into how neural networks work and how they learn data.

Structure. The first chapter will describe the initial problem in mathematical detail. To do this, we first look at various ways of storing geometric data explicitly before we go into more detail about the method using implicit functions. After that, we consider some classical approaches that solve the problem by discretizing the target space. We then turn our attention to the chosen approach via neural networks. Finally, we discuss the fundamentals of deep learning and high-level ideas on how to use them to learn single and multiple implicit functions.
In the second chapter, we lay the mathematical basis for the following chapters. To do this, we consider variational problems and how to compute their minimizers. We also consider how sequences of minimizing variational problems behave. In the last part, we then define the perimeter of closed surfaces.
In the third chapter, we formulate the variational problems for the surface reconstruction problem. First, we study the Modica-Mortola-based approach in more detail and show how a distance function can be obtained from the corresponding phase field. After that, we derive our new approach based on Ambrosio-Tortorelli. We then compare both approaches by considering a simple 1D example where we reconstruct a single point.
In the fourth chapter, we develop the framework for our numerical experiments. Therefore, we choose reasonable starting values for optimizing the neural networks and a suitable architecture for them.
In the fifth chapter, we consider the numerical implementation of the variational problems. First, we consider the rendering of the level sets and show where difficulties can arise. We then use this to construct images of the level sets of implicit functions of various geometric objects in both 2D and 3D. We also test the effects of different parameters of the variational problem and the network architecture.
In the sixth chapter, we then turn to the case where we want to train a network for several phase fields. For this, we use a new approach that uses an autoencoder to extract features from the point cloud. Then, we train this simultaneously to the phase field network using the loss functional based on Ambrosio-Tortorelli.

Chapter 1

Surface reconstruction problem

In this chapter, we will formulate the initial problem mathematically precisely. To do this, we first look at the different ways to explicitly store geometric data. Then, Section 1.2 considers the implicit approach, in which an object is described as a level-set of an appropriate function. Our goal will be to determine such functions. For this purpose, we consider various classic approaches in Section 1.3, i.e., those that do not use neural networks, which we turn to in Section 1.4. Sections 1.5 and 1.6 describe the high-level ideas for surface reconstruction and shape space learning.

1.1 Storing data

Storing geometric data, such as 3D objects, can be very challenging. There are many ways to do this, all with advantages and disadvantages. We want to explore some examples of them based on [Pou20]. The most intuitive ones are **explicit** representations. These are the ones where we can directly obtain the precise location of the objects from the definition.
One way of representing a shape is by using a **parameterization**. This is a continuous form of representation, where the objects are given as the image of a function. For example, if we consider the circle with origin in 0 and radius 1, the map

$$t \mapsto \begin{pmatrix} \sin(t) \\ \cos(t) \end{pmatrix} \tag{1.1}$$

gives a parametrization for this circle. However, if we want to be able to consider more complex objects, finding a parametrization is a challenging task, and not all shapes can be described by one alone. Therefore we start considering discrete ways of representing objects. There are four major types of them.

- **Point clouds.** Point clouds are unordered sets of points, where every point is a vector consisting of different features. Examples of features are the points coordinates, the color value, or the corresponding normal vector at this position. Point clouds' advantage is that this representation is closest to the raw data output from sensors, such as depth cameras or laser scanners. They work by analyzing the reflected light emitted by a laser or LED source, see [HHEM20]. In these cases, the points are sampled from the surface of a geometrical object. There is no need for further assumptions on the points distribution on the surface for the sampling process. However, we run into difficulties if we want to

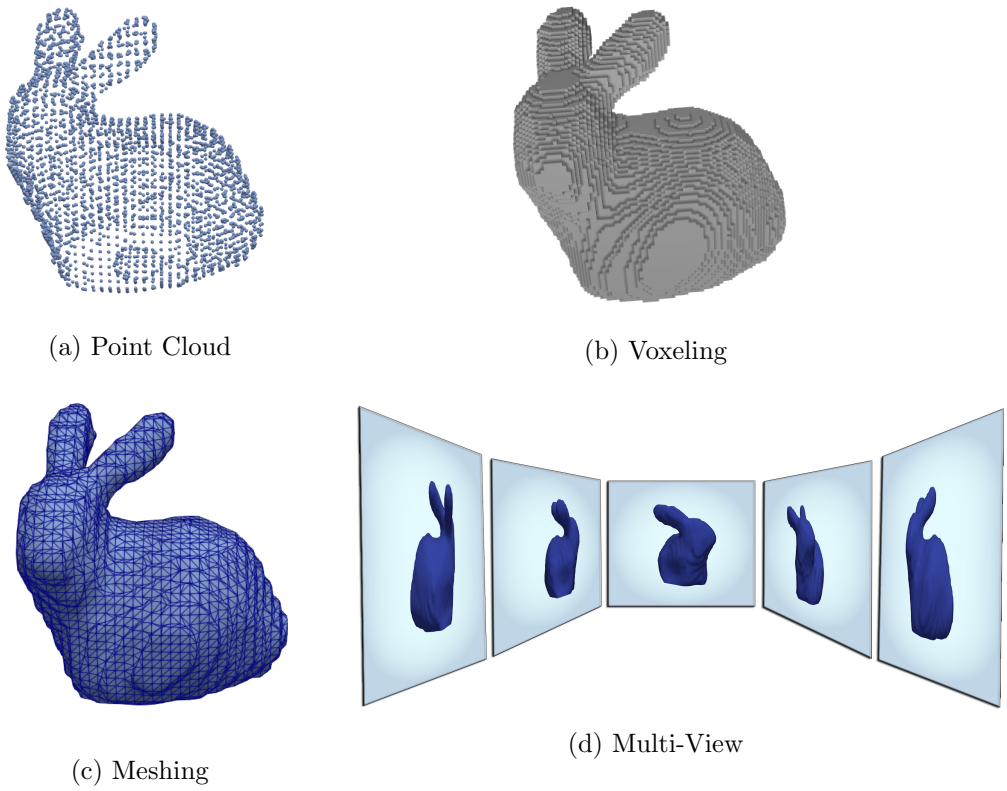
visualize them. Rendering the bare point cloud is easy, but we are frequently much more interested in rebuilding the original object. Furthermore, since point clouds do not contain much information about the geometry or topology of the underlying objects, they do not produce water-tight surfaces right away. So before rendering, it makes sense to convert the point cloud into one of the other representation formats. Another problem with point clouds is that they need much storage for high-resolution objects.

- **Voxeling.** If we want to represent data in the 2D case, we often use images as representation. Images can be seen as matrices, where every entry depicts a same-sized square called a pixel. Voxeling is the technique of doing the same one dimension higher. Therefore, we now consider a three-dimensional tensor, where every entry describes the properties of a particular cube. The advantage of this representation is that it gives us a vector space structure, which allows us to perform mathematical operations on the data, like addition or convolution. For implementing the voxeling approach, one needs to specify the resolution, which determines the size of the individual cubes. If we choose the size too small due to the cubical growth of the number of cubes, we will have to expect a very high memory requirement, even if we only deal with spatial data. On the other hand, if we choose the cube size too big, we get too pixelated results, which can lead to a loss of information and does not look very natural.
  Today the primary use of voxels is to handle volumetric data. This data type often occurs in medicine, for example, in magnetic resonance imaging (MRI), see [HWS+08]. In this case, the matrix entries are scalars describing the attenuation of X-ray radiation by different types of tissues.

- **Meshing.** Another way of producing surfaces is by using meshes. A mesh is an approximation of the object with simple small geometric figures. It consists of vertices that are points sampled from the surface of the underlying object and edges connecting nearby points constructing faces of a polygonal shape. In 3D, these shapes are mainly triangles since any three points can define the vertices of a triangle as long as they are linearly independent. Since the sizes of the different triangles in the mesh do not need to be the same, we get a better approximation, yielding in more natural-looking visualization. Therefore, artists often use meshes to create 3D models for animated movies. Meshes are also straightforward to process and have low memory requirements.
  The difference to the voxeling approach is that meshes can only be used to represent the surfaces of objects but are not suitable for solid objects.
  To compute a mesh, one needs to find a proper algorithm to find a valid triangulation for points. Since triangulations are not unique, a possible choice is where the minimal interior angle over all triangles gets maximized, see [Del34]. An equivalent description of this triangulation is that the circumcircle of each mesh triangle cannot contain any other point. One possible algorithm to compute this can be found in [Bow81].

- **Multi-View.** The last approach can only be used for 3D data. The idea is to take images from the object from different angles and reduce the 3D data problem by one dimension to a 2D problem. The viewpoints from where the images have been taken are either known or calculated from the data. One significant advantage of this representation is that, in

practice, this data type is straightforward to get at a low cost.
Other benefits of this method are, for example, that it handles well the effect of noise and incompleteness. However, it is still an open question of how many images are sufficient to capture the whole object. For example, it takes way more images to represent an object containing many cavities than one with a smooth surface.



(a) Point Cloud

(b) Voxeling

(c) Meshing

(d) Multi-View

Figure 1.1: The four explicit ways of storing data

Instead of using these representations, we will focus on a less visual representation for data, the **implicit representation**. Implicit means we will consider the given data as the level set of a scalar-valued function. For example, we can denote the unit circle as the zero-level set of the function

$$(x, y) \mapsto x^2 + y^2 - 1 \tag{1.2}$$

for $(x, y) \in \mathbb{R}^2$. The ground truth function for implicit representation can describe any level of detail of the object. Through discretization, this function gets approximated by functions of a finite-dimensional vector space. The question we are dealing with here is, given data in an explicit representation, how can one find a function for the implicit representation?
Since it is straightforward to convert a voxeling or meshing representation into a point cloud by simply sampling points from the surface, we will use point clouds as the starting point for all of our computations. There is also ongoing work to extract implicit functions from image view input, for example, [HLC+18, MST+20, YKM+20], but we will not focus on that.

1.2 Implicit representations

Suppose we have given a bounded space $\Omega \subseteq \mathbb{R}^d$ and a closed surface $\mathcal{S} \subseteq \Omega$ and from this surface we sample a point cloud $\mathcal{P} = \{p_1, \dots, p_n\}$. For now, we will only consider the coordinates as features of the point cloud, i.e., $\mathcal{P}$ is a subset of Ω. Now our goal will be, given this point cloud, to reconstruct the original surface $\mathcal{S}$. As mentioned before, we want to find an implicit representation for $\mathcal{S}$. That means, given only $\mathcal{P}$, we want to find a function $u\colon\ \Omega \to [-1, 1]$, such that it holds

$$\mathcal{S} = \{x \in \Omega \mid u(x) = 0\}. \tag{1.3}$$

Note that this representation also gives us a disjoint division of our space Ω into three parts: our surface $\mathcal{S}$, the interior of the surface

$$\mathcal{I} := \{x \in \Omega \mid u(x) < 0\} \tag{1.4}$$

and the exterior

$$\mathcal{O} := \{x \in \Omega \mid u(x) > 0\}. \tag{1.5}$$

If u is a solution to the surface reconstruction problem, we can easily see that $-u$ is also one. So the choice of interior and exterior is arbitrary. This shows that the problem is not uniquely solvable and, therefore, ill-posed.

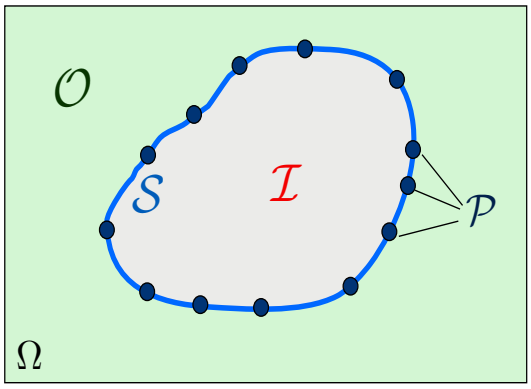


Figure 1.2: A 2D-example for the setting of surface reconstruction

For the function u, we only care about its zero-crossings and not so much its behavior outside of $\mathcal{S}$. That means we obtain a degree of freedom based on what the function does elsewhere. Hence, we differentiate four major classes of functions that we could use for the implicit representation.

- **Indicator functions.** The easiest way of describing $\mathcal{S}$ is by the indicator function of its complement $u_{IF} = \mathbb{1}_{S^c}$. This function is 0 if a point lies on the surface and is 1 else-wise. If we use this for the surface reconstruction problem, the interior will always be the empty set. Therefore, this approach is valuable for describing open surfaces.

- **Occupancy functions.** These are all the functions that can be written as

$$u_{OF}(x) := \begin{cases} 1 & x \text{ is inside} \\ 0 & x \text{ is on the boundary} \\ -1 & x \text{ is outside} \end{cases}. \tag{1.6}$$

The image of these functions is also discrete, and they can be seen as a more general version of indicator functions.

- **Distance functions.** These functions also encode the distance between the given point and the original surface $\mathcal{S}$. Assuming d is a metric on Ω, then a distance function is given by

$$u_{DF}(x) := d(x, \mathcal{S}) = \inf_{y \in \mathcal{S}} d(x, y). \tag{1.7}$$

Assuming that $\mathcal{S}$ is closed, the infimum is a minimum. In the special case that d is the discrete metric, this is equivalent to the indicator function approach. While we could use any metric for the reconstruction, in the following, we refer to distance functions as the ones that arise from the Euclidean distance. The reason is that these functions have significant properties.

Lemma 1.1:
(Chapter 2.1.2 in [Sig06]): Let u_{DF} be a distance function. Then for almost every $x \in \Omega$ it holds

$$\|\nabla u_{DF}(x)\| = 1. \tag{1.8}$$

Note that (1.8) can not be fulfilled everywhere since there are points where the distance function is not even differentiable. These irregularities are called the **cut locus** of the level set. Formally, they are defined as the closure of all those points for whom there are at least two different shortest paths to the boundary. One can show that the complement of the cut locus is the maximal open subset of the complement of the level set, where the distance function is continuously differentiable, see [Wol95]. If the boundary is a smooth manifold, the parts of the cut locus that lie within the interior are called **medial axis** of the object. They can be equivalent, characterized as the union of all maximal discs' center points that fit within the interior. Maximal, in this case, means that no disc with a larger radius in the interior exists that contains that disk. For example, in the case of a circle, the cut locus contains only the center point. For a square, the medial axes are the diagonal axes.

- **Signed distance functions (SDF).** If we have given an occupancy function u_{OF} we get a signed distance function u_{SDF} by

$$u_{SDF}(x) := u_{OF}(x) \cdot d(x, \mathcal{S}) \tag{1.9}$$

where $d(x, \mathcal{S})$ is the distance from the point x to $\mathcal{S}$. A signed distance function maps every

point to its positive distance to $\mathcal{S}$ if the point is inside and the negative distance if it is outside. They can be seen as a continuous version of the occupancy functions. Note that given a signed distance function, it is straightforward to compute an occupancy function. However, the other round way is challenging if we do not know $\mathcal{S}$ in advance.
One substantial advantage is that one obtains further information about the shape by considering the derivatives of the signed distance function in the points where the function is differentiable. For example, the gradient of the representing function ∇u_{SDF} equals the outward-oriented surface normal $n_{\mathcal{S}}$ of the level set $\mathcal{S}$. From this, we can also compute the projection of the tangent plane in a point by

$$P_T(x) = \mathbb{1} - n_{\mathcal{S}}(x) n_{\mathcal{S}}(x)^T. \tag{1.10}$$

The second derivative yields information on the curvature of the shape. For example, one can define the mean curvature as the average of the trace of the Hessian matrix or the Gaussian curvature as the determinant of the Hessian matrix. These curvatures are useful for manipulating the shape by deformations; see [IRS17].

One benefit of using one of these functions is that it is elementary to combine them, see [Ric73]. For example, if we consider two objects A and B, that are given by their defining signed distance functions $s(A)$ and $s(B)$, then it holds

$$\tilde{s}(A \cap B) = \max\{s(A), s(B)\} \qquad \text{and} \qquad \tilde{s}(A \cup B) = \min\{s(A), s(B)\}. \tag{1.11}$$

Here $\tilde{s}(A \cap B), \tilde{s}(A \cup B)$ give an implicit representation of the intersection and union of the objects but are not necessarily signed distance functions. In general, computing a signed distance function is very challenging, but there are some objects for which it is well known. For example, for the unit circle, the implicit representation given in (1.2) is a signed distance function. That means if $\mathcal{S}$ can be expressed by the union or intersection of simple geometric objects, we now have a way of computing their defining function. However, for arbitrary objects, we need to take a different approach.

1.3 Classical approaches to surface reconstruction problem

We want to start by discussing some of the "classical" approaches for solving the surface reconstruction problem, which are the ones that give a direct construction of u. Their idea is to consider a finitely generated vector space $\mathcal{V} := \operatorname{span}\{\phi_i \mid i = 1, \dots, n\}$ and to write

$$u = \sum_{i=1}^{n} \alpha_i \phi_i \tag{1.12}$$

as a linear combination of the basis elements. We briefly want to take a look at two examples.

The first approach we want to look at is from [CBC+01] using signed distance functions. Here we formulate the surface reconstruction problem as an interpolation problem. We do not only interpolate at the points of the point cloud because then the function that is constant 0 would

be a solution, but we use some additional points that we obtain from going from the original points in the direction of the normal vector. The idea is that if we make the steps small enough, we end up in the interior/exterior and know the distance from this point to the surface. So, for some $\varepsilon > 0$ want to interpolate the points

$$u(p_i) = 0 \qquad u(p_i + \varepsilon n_i) = -\varepsilon \qquad u(p_i - \varepsilon n_i) = \varepsilon \qquad \forall i = 1, \dots, n \tag{1.13}$$

where n_i is the outgoing normal vector in p_i given in advance. The interpolant should represent a signed distance function close to the surface. There are many ways how one can solve interpolation problems. One strategy is to use polynomials. The downside is that if we use high-order polynomials, the functions oscillate heavily between the points. This behavior is called Runge's phenomenon, see [Run01]. Another approach for interpolation problems is by using radial basis functions. These are all the functions whose values only depend on the distance of the input to the origin. One examples would be $\Phi(x) = \|x\|$. The problem with these functions is that the results degenerate in the extrapolation, which is the function's behavior away from the target points. In practice, one gets the best results by using a mixture of radial basis functions and a low-degree polynomial

$$u(x) = \sum_{i=1}^{M} \alpha_i x^i + \sum_{i=1}^{n} \beta_i \Phi\left(x - x_i\right), \tag{1.14}$$

where $\{x_i\} = \bigcup_i \{p_i, p_i \pm \varepsilon n_i\}$ are the points to be interpolated. In order to estimate the corresponding coefficients α_i and β_i, we insert (1.14) into the interpolation conditions and end up with a system of linear equations that we then need to solve.
In [ZZW19], one can find a swift implementation of this method. One way of improving the radial basis function ansatz can be found in [TO02]. They do not solve the interpolation problem but only approximate the function values. Therefore the results are much better when working with noisy data.

The second approach is the Poisson surface reconstruction from [MK06]. We want to solve the surface reconstruction problem via an occupancy function. The idea is that the gradient of the occupancy function is almost everywhere zero but on the surface. Therefore we start by extending the given normal vectors to a vector field $V \colon \Omega \to \mathbb{R}^3$. Since we want the gradient of the function we are looking for to be similar to V, we consider the minimization problem

$$\min_u \int \|\nabla u(\xi) - V(\xi)\|^2 \, \mathrm{d}\xi. \tag{1.15}$$

Using the Euler Lagrange formula, which we will discuss later, one can show that it is enough to solve $\Delta u = \nabla \cdot V$, see (2.21). We solve this partial differential equation using finite elements. Using (1.12), with ϕ being spline functions, we get the following linear system of equations that we know how to solve

$$\sum_{j=1}^{n} \alpha_j \int \langle \nabla \phi_i(\xi), \nabla \phi_j(\xi) \rangle \, \mathrm{d}\xi = \int \langle \nabla \phi_i(\xi), V(\xi) \rangle \, \mathrm{d}\xi \qquad \forall j = 1, \dots, n. \tag{1.16}$$

This approach was later improved in [KH13] by adding a regularization term to the minimization problem

$$\min_u \int \|\nabla u(\xi) - V(\xi)\|^2 \, \mathrm{d}\xi + \lambda \sum_{p \in \mathcal{P}} |u(p)|^2 \tag{1.17}$$

that encourages the function to be zero in the original points. The resulting system of linear equations is then

$$\sum_{j=1}^{n} \alpha_j \left(\int \langle \nabla \phi_i(\xi), \nabla \phi_j(\xi) \rangle \, \mathrm{d}\xi + \lambda \sum_{p \in \mathcal{P}} \phi_i(p) \phi_j(p) \right) = \int \langle \nabla \phi_i(\xi), V(\xi) \rangle \, \mathrm{d}\xi \quad \forall j = 1, \ldots, n. \tag{1.18}$$

These attempts all have in common that for good results, even on small point clouds, we will need to know normal vectors in advance and always end up with a large system of linear equations. Another famous approach for solving the surface reconstruction problem with indicator functions is from [ACSTD07], where they use the Voronoi diagram of the input points to approximate the surface's normals.
The approach we want to focus on is very different. Instead of explicitly constructing the function, we want to "learn" it using neural networks. The difference from the classic approaches is that neural networks provide a different spatial discretization that no longer depends on the target space.

1.4 Deep neural networks

Before we explain how we want to learn occupancy or signed distance functions, we want to start with a broad overview of artificial intelligence based on the first chapter in [Agg18]. In general artificial intelligence is a subsection of computer science and mathematics. It deals with machines mimicking functions associated with human cognition. We consider it as the attempt of a computer to recreate specific decision-making structures of the human brain.
The techniques and processes used for solving these problems are summarized under the term **machine learning**. Instead of explicitly solving a problem, we put all the data in a generic algorithm that can solve the problem. So machine learning analyses data to learn and then use this knowledge to make decisions. We differentiate various categories based on how the machine learning algorithm performs the learning.
For our work, we mainly focus on **supervised learning** problems. These are the problems where an algorithm should approximate a mapping function that predicts the output for a given observation. We train the algorithm with data from which we already know the desired outputs. Output variables of the training data are often called labels or categories. It is common for classification models to predict continuous values as the probabilities of a given example belonging to each output class. The final prediction is made by selecting the class label with the highest probability. The algorithm should adjust itself to minimize the error between the predicted output and the correct labels. Suppose we want to train an AI that recognizes advertising in E-mails. In this case, the possible labels for an E-mail are either "spam" or "no spam". So we

train the algorithm by feeding it with E-Mails from which we already know whether they are spam or not, and in the end, the program should get a new mail and decide on its own.
Since the late 2000s, **deep learning** has been the most successful approach to supervised learning problems. It uses **artificial neural networks** to solve the problems we stated before. Any neural network consists of the same three parts: the inputs form an input layer, the middle layers which perform the processing of the inputs are called hidden layers, and the output forms the output layer. The hidden layers consist of **neurons** that perform a mathematical operation on their associated inputs. Each input to a neuron is associated with a different weight, which affects the computation. So neural networks represent functions of their given input by propagating the computed values from one layer to the next until they reach the final output layer. The key to neural networks is adjusting the weights so that the function it represents solves the initial problem. We can think of artificial neural networks as universal function approximators.
Now we consider the easiest case of an ANN, a **multi-layer perceptron (MLP)**. MLPs are the most widely used architecture for neural networks. To understand how they work, we look at what happens in the neurons of the MLP. The neurons do two important things. Foremost, they calculate a weighted sum of their incoming data by performing an affine linear transformation to it. Afterwards the transformation is inserted into an **activation function** σ. A common choice for an activation function is the **Rectified Linear Unit (ReLU)** $\sigma(x) := \max\{0, x\}$. This function is the identity for all positive inputs and maps all negative values to 0. So if we consider a single neuron that gets n inputs x_i and is associated with weights w_i, the output of this neuron would be

$$\sigma\left(\sum_{i=1}^{n} w_i x_i + w_{n+1}\right). \tag{1.19}$$

For evaluating an MLP, we repeat this calculation for every neuron until we reach the output layer.

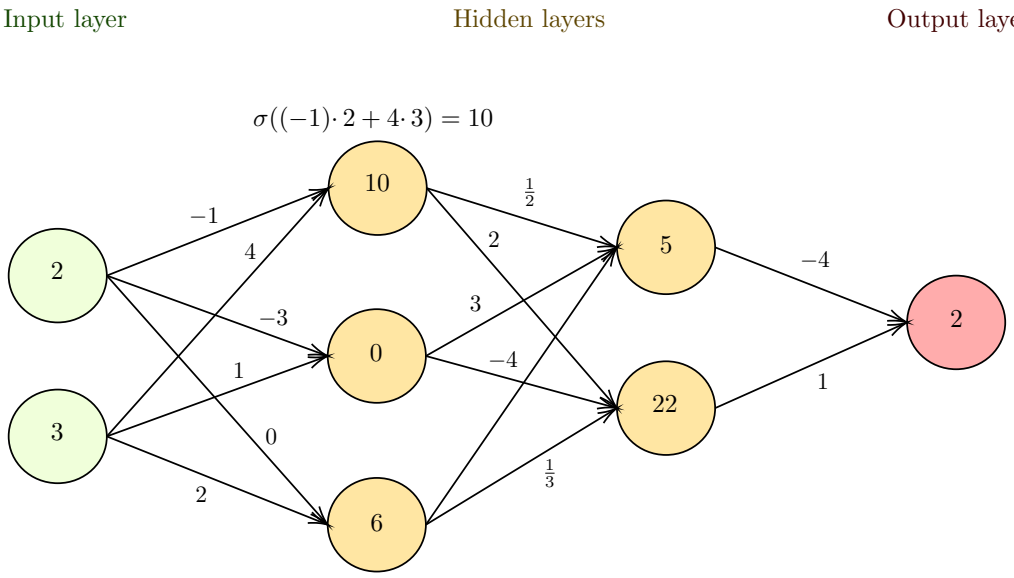


Figure 1.3: example of the feed-forward of a neuronal network with two hidden layers, ReLU activation function, and no bias

Now we want to use the MLP to solve supervised learning problems. Let u_θ the by function

representing the architecture of the network, θ be the weights of the network summarized as a vector, $(x_i)_{i=1}^n$ be the training data and $(y_i)_{i=1}^n$ be the corresponding labels. For the training of the MLP, we need to measure how good the network's predictions are. To this end, we choose a **loss functional** $\mathcal{L}$. This is a map that takes the predicted labels and the actual labels as input and returns a value measuring the total error. The choice of the loss functional is crucial in what the network should learn. One example of a loss function is the mean squared error between the predicted and target labels. Another possibility is to choose $\mathcal{L}$ to be a partial differential operator. In this case, the loss is the sum of a PDE residual and the deviation from the boundary condition. The resulting networks are called **physics-informed neural networks** (PINN), see [CdCG+22]. After training, these networks should give an approximation of the solution to the equation. Rather than deducing a solution based only on data, they bear in mind the physics of their underlying partial differential equation.

During the training process, the network parameters change until the total error given by the loss functional gets minimized. Therefore, we can write the supervised learning problem using the loss function as the following minimization problem

$$\min_{\theta} \mathcal{L}\big((u_\theta(\vec{x}_i))_{i=1}^n , (y_i)_{i=1}^n \big). \tag{1.20}$$

Many numerical algorithms exist to solve such a minimization problem. That means we have reformulated the supervised learning problem into a problem that we understand and therefore found a solution to the original problem.

We now want to specify more about what a solution is. By increasing the number of layers and neurons, the network can learn more complex functions on the training set. That means if we have a too small network, we will not be able to acquire good results. However, making the network too big will provide good performance on the training data but not guarantee good performance on unseen test data. This problem is called **overfitting**. It arises if the network learns a function that matches is too specific to the underlying training data. It means the network learns the correct relationship between each input and output value, not the entire pattern.

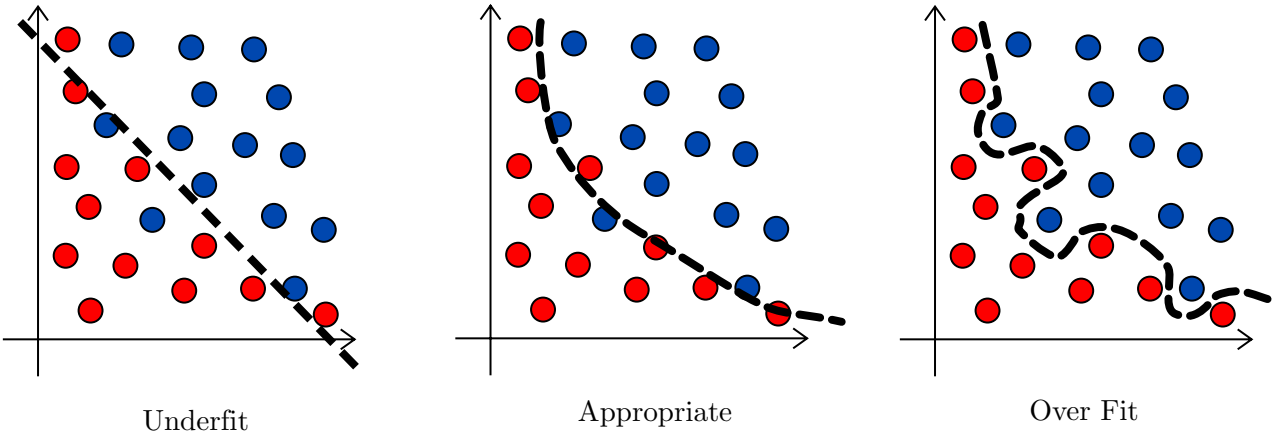


Figure 1.4: 2D example of under/over-fitting on a binary classification problem

One way of reducing the overfitting error is by adding a regularization term to the loss functional that penalizes excessive fluctuations of the network function. For some $\alpha > 0$, the final loss is

$$\mathcal{L} = \mathcal{L}_{data} + \alpha \mathcal{L}_{reg}. \tag{1.21}$$

Here, $\mathcal{L}_{data}$ is the loss between the predicted labels and the actual labels, and $\mathcal{L}_{reg}$ is the regularization term. If α is too small, the regularization has no effect, and overfitting can still happen. If α is too large, the training data and target label lose influence on the loss function. So we are dealing with a trade-off between the data error and the approximation error of the regularization. An appropriate solution function should map the inputs close to their labels and remain as simple as possible. We will later use regularization to control the evenness of the objective function.

1.5 Learning implicit surface representations

Now we want to get back to the surface reconstruction problem. The idea is to consider it a supervised learning problem, with the training data being the point cloud since we know how our function should behave there. That means the training data are the points $p_i \in \mathcal{P}$, and the corresponding labels are $y_i = 0$.
As the neural network u_θ for the implicit representation of $\mathcal{S}$, we choose an MLP. The network should take a single point as input; therefore, we need d inputs for the input layer. The output layer should be one-dimensional. The only thing that remains to specify now is the loss functional. That means we can reformulate the surface reconstruction problem from a high-level perspective as follows:

$$\text{Find a loss functional } \mathcal{L} \text{ such that } S = \{x \in \Omega \mid u_{\theta_*}(x) = 0\} \text{ with } \theta_* = \arg\min_\theta \mathcal{L}\big(u_\theta, \mathcal{P}\big).$$

Next, we want to eliminate the weights in the problem formulation. Therefore we need the following result for multi-layer perceptrons. It explains why neural networks are suitable for approximating unknown functions.

Theorem 1.2 (Universal approximation theorem):
(Theorem 1 in [Han19]): Let $\Omega \subset \mathbb{R}^d$ be compact. For any map $u : \Omega \to \mathbb{R}$ continuous, there is a sequence of functions $(u_n)_{n\in\mathbb{N}}$ representing a MLP with ReLU activation functions and $d + 2$ hidden layers, such that

$$\lim_{n\to\infty} \|u - u_n\|_{C_0(\Omega)} = 0. \tag{1.22}$$

The theorem does not provide any convergence rates, i.e., it yields no information on how large the network has to be to achieve minor errors. However, the big takeaway is that we can approximate any continuous function with a neural network. Therefore, we no longer need to consider the supervised learning problem as a minimization problem over some weights but over the whole space of functions to find a suitable loss function. That means it is sufficient to solve:

$$\text{Find a loss functional } \mathcal{L} \text{ such that } \mathcal{S} = \{x \in \Omega \mid u_*(x) = 0\} \text{ with } u_* = \arg\min_u \mathcal{L}\big(u, \mathcal{P}\big).$$

Since we will need large networks for this, we must address the overfitting problem. So let $\mathcal{L} = \mathcal{L}_{data} + \alpha\mathcal{L}_{reg}$ as before. The data loss is responsible for making the function 0 along $\mathcal{S}$. The regularization error is responsible for making the function as smooth as possible elsewhere and punishing high gradient values. In this thesis, we are going to discuss several choices for losses.

1.6 Learning shape spaces

The approach of using deep neural networks for surface representations originates in a different but closely related area of computer vision, which is the problem of shape space learning, see [PFS+19]. The idea is again to get an implicit representation of a surface from a point cloud by using a neural network. While in the surface reconstruction problem, we train a neural network on a single shape, in the shape space learning setting, we train a single network on multiple shapes. Therefore the model can represent a wide variety of entities. To this end, the network gets an additional input representing the entirety of the object. Ultimately, we want these shapes to have an implicit representation given by a function $u^{(i)}$. Formally, this means if we have a set of point clouds $\mathcal{P}_i$ that were sampled from shapes

$$\mathcal{S}_i = \left\{ x \in \Omega \mid u^{(i)}(x) = 0 \right\} \supset \mathcal{P}_i \tag{1.23}$$

then the trained network should approximate $u_\theta(x, \mathcal{P}_i) \approx u^{(i)}(x)$.
We do not want to enter the point cloud directly into the network function since many points would result in many parameters needed. To this end, we first map it to a low-dimensional vector space called latent space or feature space. This mapping should be continuous and without the loss of much information. Therefore, we need the shapes $\mathcal{S}_i$ to be part of a submanifold with a dimension lower than the dimension of the latent space. After computing the resulting latent vector/feature vector z_i from point cloud $\mathcal{P}_i$, we concatenate it to a point $x \in \Omega$ and use this as an input for the shape space learning network.
There are many ways to obtain a feature vector from a given shape. Once we get the feature map, we still need a loss function for training the neural network, that is

> Find a loss functional $\mathfrak{L}$ such that $\mathcal{S}_i = \{x \in \Omega \mid u_{\theta_*}(x, z_i) = 0\}$ with $\theta_* = \arg\min_\theta \mathfrak{L}\big(u_\theta, \{\mathcal{P}_i\}_i\big)$.

The easiest way of handling this problem is by reducing it to the surface reconstruction problem. Therefore assume that we already have given a loss functional $\mathcal{L}$ for the surface reconstruction problem. Then we can define a loss function for the shape space learning by

$$\mathfrak{L}\big(u_\theta, \{\mathcal{P}_i\}_i\big) := \sum_i \mathcal{L}\big(u_\theta(\cdot, z_i), \mathcal{P}_i\big) \tag{1.24}$$

The only thing that is left is how to find a good feature vector. One way is to obtain optimal feature vectors regarding the loss functional by making them learnable parameters for the neuronal network. That means while training a neural network f with network parameter θ, we

solve

$$z_i^*, \theta_* = \arg\min_{z_i, \theta} \sum_i \mathcal{L}\big(u_\theta(\cdot, z_i), \mathcal{P}_i\big) \tag{1.25}$$

Once the training is done, we can extract a feature vector from a single point cloud $\mathcal{P}$ by optimizing in the latent space, that is

$$z_* := \arg\min_z \mathcal{L}\big(u_{\theta_*}(\cdot, z), \mathcal{P}\big). \tag{1.26}$$

The problem with this is that every time we want to process a new shape of the submanifold, we need to do a whole optimization process. We will later see a way of getting feature vectors directly and not as solutions of a minimization.

In one of the first papers on this topic by Park et al., the training for a single shape is seen as "neither feasible nor useful" (Chapter 4, [PFS+19]). In the following chapters, we will focus primarily on the surface reconstruction problem and show that this problem is indeed solvable and has many valuable applications. Later we use the results for shape space learning while using another neural network to obtain the features from the point cloud.

Chapter 2

Mathematical prerequisites

In this chapter, we lay the mathematical foundations for the following chapters. The first concept we consider is Γ-convergence, which is needed to study sequences of functionals and their limit behavior. Then, in Section 2.2, we formulate the Euler-Lagrange formula that characterizes the local minima of functionals. In the last part, we then define the perimeter of an object and analyze its properties.

2.1 Gamma convergence

In the context of machine learning, our goal is always to find a minimizer for a given functional. In this chapter, whose first part is based on [Bra02], we want to find reasonable assumptions, like continuity, to take on the functional to ensure the existence of minimizers and convergence. The functionals we consider map from infinite-dimensional Banach spaces X into the extended real line. However, a function that attains the value ∞ can not be continuous. Also, continuity is far too restrictive and, for our purposes, not even necessary. Therefore, we introduce a weaker form.

Definition 2.1:
A functional $\mathcal{F}\colon X \to \overline{\mathbb{R}}$ is called **lower semi-continuous** (resp. weak lower semi-continuous) **in** u, if for every sequence $u_k \to u$ (resp. $u_k \rightharpoonup u$) holds

$$\mathcal{F}(u) \leq \liminf_{k\to\infty} \mathcal{F}(u_k). \tag{2.1}$$

A functional is called **lower semi-continuous** (resp. weak lower semi-continuous), if it is lower semi-continuous (resp. weak lower semi-continuous) in every $u \in X$.

As an example, consider the functional $\mathcal{F}\colon \mathbb{R} \to \mathbb{R}$, with

$$\mathcal{F}(x) = \begin{cases} 1 & \text{if } x \neq 0 \\ \alpha & \text{if } x = 0 \end{cases}. \tag{2.2}$$

This is lower semi-continuous, if and only if $\alpha \leq 0$. Note that weak lower semi-continuity is a stronger assumption than lower semi-continuity. Now one can show the existence of a minimizer if we have additional information on the functional.

Lemma 2.2:
(Theorem 2.1 in [Mut10]): Let $\mathcal{F}\colon X \to \overline{\mathbb{R}}$ be lower semi-continuous, and assume that for every $\lambda \in \mathbb{R}$ the sublevel sets $\{x \in X :\ \mathcal{F}(x) \leq \lambda\}$ are compact. Then there exists a $\tilde{u}$, with

$$\mathcal{F}(\tilde{u}) = \inf_{u \in X} \mathcal{F}(u). \tag{2.3}$$

Proof: Take a minimizing sequence $(u_k)_{k\in\mathbb{N}}$ with

$$\mathcal{F}(u_k) \longrightarrow \inf_{u \in X} \mathcal{F}(u). \tag{2.4}$$

If $\inf_{u\in X} \mathcal{F}(u) = \infty$, then there is nothing to show. If the infimum is finite, there must be a $C \geq 0$, such that $\mathcal{F}(u_k) \leq C$, so by assumption, there is a convergent subsequence $(u_{k_j})_{j\in\mathbb{N}}$ with $u_{k_j} \longrightarrow \tilde{u}$. By the lower semi-continuity, we conclude

$$\inf_{u \in X} \mathcal{F}(u) \leq \mathcal{F}(\tilde{u}) \leq \liminf_{j\to\infty} \mathcal{F}(u_{k_j}) = \inf_{u \in X} \mathcal{F}(u). \tag{2.5}$$

Therefore, it holds $\inf_{u\in X} \mathcal{F}(u) = \mathcal{F}(\tilde{u})$.

□

The additional property of the last lemma is called **coercivity** of the functional.
Now we know about the existence of minimizers. However, the functional may be tough to evaluate. So instead, we want to find an approximation of this functional, that converges against it. The question is, what definition of convergence do we use here? Since we are mainly interested in minimizers, we would ideally want the approximations minimizers to converge to a minimizer of the initial functional. One way of enforcing this is to consider uniform convergence, but we will see in practice that this assumption is often too strong.
So we need a new definition of what convergence of functionals should mean. The Italian mathematician Ennio de Giorgi first developed this new definition in [DG77, DG78]. We use the definitions and proofs from [Mut10].

Definition 2.3:
Given a metric space (X, d) and a family of functionals $\mathcal{F}_n\colon\ X \to \overline{\mathbb{R}}$, we say $\mathcal{F}_n$ **$\Gamma-$converges** to a functional $\mathcal{F}\colon\ X \to \overline{\mathbb{R}}$, if for every $x \in X$ the following two properties hold:

(i) For every sequence $(x_n)_{n\in\mathbb{N}} \subset X$ with $x_n \longrightarrow x$ it holds

$$\mathcal{F}(x) \leq \liminf_{n\to\infty} \mathcal{F}_n(x_n). \tag{2.6}$$

(ii) There exists one sequence $(\tilde{x}_n)_{n\in\mathbb{N}} \subset X$ with $\tilde{x}_n \longrightarrow x$ and

$$\limsup_{n\to\infty} \mathcal{F}_n(\tilde{x}_n) \leq \mathcal{F}(x). \tag{2.7}$$

We denote the convergence as $\mathcal{F}_n \xrightarrow{\Gamma} \mathcal{F}$.

The first property in the Γ-convergence definition is often called the lim inf-property, and the

second one lim sup-property. The sequence we used in the lim sup property is called the recovery sequence. One can show that Γ-limits are lower-semicontinuous. With this new definition, we can investigate how minimizers behave under this convergence.

Lemma 2.4:
(Theorem 6.17 in [Arn20]): Let $\mathcal{F}_n \colon X \to \overline{\mathbb{R}}$ be a family of functions and let x_n be a minimizer of $\mathcal{F}_n$. Suppose that $(x_n)_{n\in\mathbb{N}}$ converges to x and $\mathcal{F}_n \xrightarrow{\Gamma} \mathcal{F}$. Then x is a minimizer of $\mathcal{F}$.

Proof: Let $y \in X$ be arbitrary. By the second property of Γ-convergence we can find a sequence (y_n) with $y_n \longrightarrow y$ and $\limsup_{n\to\infty} \mathcal{F}_n(y_n) \leq \mathcal{F}(y)$. We obtain

$$\mathcal{F}(x) \overset{(2.6)}{\leq} \liminf_{n\to\infty} \mathcal{F}_n(x_n) \leq \limsup_{n\to\infty} \mathcal{F}_n(x_n) \leq \limsup_{n\to\infty} \mathcal{F}_n(y_n) \leq \mathcal{F}(y). \tag{2.8}$$

Since y was arbitrary, we can take the minimum and get $\mathcal{F}(x) \leq \min_{y\in X} \mathcal{F}(y)$.

□

So if we have a sequence of functionals $\mathcal{F}_\varepsilon$ with corresponding minimizers u_ε and we want to find a minimizer for $\varepsilon \to 0$, we need two things: First, the $\mathcal{F}_\varepsilon$ should Γ-converge and second the u_ε need to converge in the usual sense. Showing that the sequence of minimizers does converge can be challenging since we do not want to compute them. That is why we want to find some additional properties of the functionals under which we can drop this condition.

Definition 2.5:
A sequence of functionals $F_\varepsilon : X \to \overline{\mathbb{R}}$ is called **equi-coercive** if for every $t \in \mathbb{R}$ we can find a compact set $K_t \subset X$ such that

$$\{x \in X \mid F_\varepsilon(x) \leq t\} \subset K_t. \tag{2.9}$$

Theorem 2.6:
(Theorem 3.2 in [Mut10]): Let $\{\mathcal{F}_n\}$ be a equi-coervice family of functionals and let $\mathcal{F}_n \xrightarrow{\Gamma} F$, then the minimizers of $\mathcal{F}_n$ converge and their limit is a minimizer of $\mathcal{F}$.

2.2 Euler-Lagrange formula

The next part is based on the second chapter [Arn20]. Now that we have derived a convergence theory, we want to focus more on how we can directly compute a minimizer. For example, if we consider a functional that takes a vector in $\mathbb{R}^d$ as input, we can find candidates for local extrema by checking where the derivative equals zero. Nevertheless, the functionals we mainly consider are operating on spaces of functions. The most important examples are those functionals that can be written as integrals over a function and its derivative. Still, there is a similar analogy: instead of finding the root of a function, we need to solve a partial differential equation. To derive this equation, we start by extending the definition of a directional derivative to a functional space.

Definition 2.7:
Let $h_0 > 0$ and $\mathcal{F}\colon X \to \mathbb{R}$, where X is subset of an $\mathbb{R}$-vector space. Then we define the **first variation** of $\mathcal{F}$ in u in direction $\xi \in \left\{\tilde{\xi} \mid u + h\tilde{\xi} \in X,\ \forall |h| < h_0\right\}$ as

$$\delta\mathcal{F}(u, \xi) := \psi'_{\mathcal{F},u,\xi}(0) \tag{2.10}$$

with $\psi_{\mathcal{F},u,\xi}(h) := \mathcal{F}(u + h\xi)$ for all $|h| < |h_0|$.

One can show that a product and chain rule exists for the first variation. If $\mathcal{F}$ attains a local extremum in u_0, then the real-valued function $\psi_{\mathcal{F},u,\xi}$ has a local extremum in 0, if it is differentiable. Therefore, in these cases, $\psi_{\mathcal{F},u,\xi}(h) = 0$ is a necessary condition for being in a local extrema. Note that in the scalar-valued case, that means for $\mathcal{F} \in C^1(\Omega, \mathbb{R})$ we have

$$0 = \delta\mathcal{F}(u, \xi) = \langle \nabla\mathcal{F}(u), \xi \rangle \tag{2.11}$$

for any valid direction ξ and therefore can conclude $\nabla\mathcal{F}(u) = 0$.
Now we want to calculate the first variation for functionals. We only consider **variational integrals** as functionals, which are the ones that can be written as

$$\mathcal{F}(u) = \int_\Omega F(x, u(x), \nabla u(x))\,\mathrm{d}x \tag{2.12}$$

for some function $F \in C^1(V, \mathbb{R})$ with $V \subset \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^d$. First, we must ensure that the first variation exists (Lemma 2.16 in [Arn20]). To simplify the notation, in the following main theorem, we set

$$\alpha := x, \qquad \beta := u(x) \qquad \text{and} \qquad \gamma := \nabla u(x), \tag{2.13}$$

so that we regard $\mathcal{F}(u) = \int_\Omega F(\alpha, \beta, \gamma)$. The goal is now to find a closed form for the first variation.

Theorem 2.8 (Euler-Lagrange formula):
(Theorem 2.23 in [Arn20]): Let $u \in C^2(\Omega) \cap C^1(\overline{\Omega})$ be a local minimum of $\mathcal{F}$ in U with $\frac{\partial F}{\partial \gamma_i} \in C^1(V)$ for $i = 1, \ldots, d$, that is $\delta\mathcal{F}(u, \xi) = 0$ for all $\xi \in C_0^\infty(\Omega)$. Then u fulfills the following partial differential equation almost everywhere

$$\frac{\partial F}{\partial \beta} - \nabla \cdot \left(\frac{\partial F}{\partial \gamma}\right) = 0. \tag{2.14}$$

Proof: We start by explicitly calculating the first variation.

$$\begin{aligned}
\delta\mathcal{F}(u,\xi) &= \frac{\partial}{\partial h}\mathcal{F}(u+h\xi)\bigg|_{h=0} \\
&= \frac{\partial}{\partial h}\int_\Omega F\left(x, u(x)+h\xi(x), \nabla u(x)+h\xi(x)\right)\,\mathrm{d}x\bigg|_{h=0} \\
&= \int_\Omega \frac{\partial}{\partial \beta}F\left(x, u(x)+h\xi(x), \nabla u(x)+h\xi(x)\right)\xi(t) \\
&\qquad + \sum_{i=1}^{d}\frac{\partial}{\partial \gamma_i}F\left(x, u(x)+h\xi(x), \nabla u(x)+h\xi(x)\right)\frac{\partial \xi}{\partial x_i}(x)\,\mathrm{d}x\bigg|_{h=0} \\
&= \int_\Omega \frac{\partial}{\partial \beta}F\left(x, u(x), \nabla u(x)\right)\xi(t) + \sum_{i=1}^{d}\frac{\partial}{\partial \gamma_i}F\left(x, u(x), \nabla u(x)\right)\frac{\partial \xi}{\partial x_i}(x)\,\mathrm{d}x
\end{aligned} \tag{2.15}$$

Here, we used the dominated convergence theorem in the third line. To shorten the notation, we drop the argument of the function. Then using the product rule for derivatives and the Gaussian formula we obtain from the previous formula

$$\begin{aligned}
0 &= \delta\mathcal{F}(u,\xi) \\
&= \int_\Omega \frac{\partial F}{\partial \beta}\xi + \sum_{i=1}^{d}\frac{\partial F}{\partial \gamma_i}\frac{\partial \xi}{\partial x_i}\,\mathrm{d}x \\
&= \int_\Omega \frac{\partial F}{\partial \beta}\xi + \sum_{i=1}^{d}\left(\frac{\partial}{\partial x_i}\left(\xi\frac{\partial F}{\partial \gamma_i}\right) - \xi\frac{\partial}{\partial x_i}\frac{\partial F}{\partial \gamma_i}\right)\,\mathrm{d}x \\
&= \int_\Omega \left(\frac{\partial F}{\partial \beta} - \nabla\cdot\left(\frac{\partial F}{\partial \gamma}\right)\right)\xi + \nabla\cdot\left(\xi\frac{\partial F}{\partial \gamma}\right)\,\mathrm{d}x \\
&= \int_\Omega \left(\frac{\partial F}{\partial \beta} - \nabla\cdot\left(\frac{\partial F}{\partial \gamma}\right)\right)\xi\,\mathrm{d}x + \int_{\partial\Omega}\left\langle \underbrace{\xi(x)}_{=0}\frac{\partial F}{\partial \gamma}, n(x)\right\rangle\,\mathrm{d}\mathcal{H}^{d-1}(x) \\
&= \int_\Omega \left(\frac{\partial F}{\partial \beta} - \nabla\cdot\left(\frac{\partial F}{\partial \gamma}\right)\right)\xi\,\mathrm{d}x.
\end{aligned} \tag{2.16}$$

Since $\xi \in C_0^\infty$ was arbitrary, the result follows from the fundamental theorem of calculus.

□

The Euler-Lagrange equation gives us a way of solving variational minimization problems. By solving the partial differential equation, we get candidates for minimizers. In the following, we consider two examples of this. The first one is from Chapter 13.2 in [MV99]. We want to formally show that the shortest path between two points a and b in the plane is given by a straight line. Therefore, we consider the corresponding minimization problem

$$\min_u \int_0^1 \sqrt{1+u'(x)}\,\mathrm{d}x \tag{2.17}$$

under the constraint $u(0) = a$ and $u(1) = b$. Using the Euler-Lagrange equation with $F(\alpha, \beta, \gamma) = \sqrt{1+\gamma^2}$ we get

$$-\frac{\mathrm{d}}{\mathrm{d}x}\frac{u'(x)}{\sqrt{1+u'(x)}} = 0. \tag{2.18}$$

If the derivative of a function is zero, then this function must be constant, hence

$$c := \frac{u'(x)}{\sqrt{1+u'(x)}}. \tag{2.19}$$

From this we conclude, that $u'(x) = \pm\sqrt{c^2/(1-c^2)}$ is also constant and therefore u has to be linear.

As the second example, we consider again the one from (1.15). Here in the variational setting $\min_u \int \|\nabla u(\xi) - V(\xi)\|^2 \,\mathrm{d}\xi$, the integrand is given by

$$F(\alpha, \beta, \gamma) = \|\gamma - V\|^2. \tag{2.20}$$

Applying the Euler-Lagrange formula yields that the minimizer u fulfills

$$\begin{aligned} & \frac{\partial}{\partial\beta}\|\gamma - V\|^2 - \nabla \cdot \left(\frac{\mathrm{d}F}{\mathrm{d}\gamma}\|\gamma - V\|^2\right) = 0 \\ \Rightarrow \quad & \nabla \cdot 2(\gamma - V) = 0 \\ \Rightarrow \quad & \Delta u = \nabla \cdot \nabla u = \nabla \cdot \gamma = \nabla \cdot V. \end{aligned} \tag{2.21}$$

2.3 Bounded variation and surface perimeter

To formally define the perimeter of a set, we need to extend our definition of differentiability to a larger class of functions for the variational case. One way of doing this is using Sobolev spaces and weakly differentiable functions. However, this is not enough for our purposes since we want to define differentiability for indicator functions, which are not contained in Sobolev spaces, since they are not continuous. Therefore we need a different approach, based on the first chapter in [Giu84].

Definition 2.9:
Let $\Omega \subset \mathbb{R}^d$ be open and $f \in L^1(\Omega)$. We define the **total variation** of f as

$$\int_\Omega |Df| := \sup_{\substack{g \in C_0^1(\Omega, \mathbb{R}^d) \\ \|g\|_\infty \leq 1}} \int_\Omega f(x)\mathrm{div}(g)(x)\,\mathrm{d}x \tag{2.22}$$

If $\int_\Omega |Df| < \infty$ we say f has bounded variation, and we denote the space of all functions with bounded variation as $BV(\Omega)$.

If f is a continuous function of one variable, the variation equals the length of the movement along the y-axis. Note that $\int_\Omega |Df|$ is just some notation and does not need that f is differentiable

in the classical sense but can be applied to a much bigger group of functions. Nevertheless, this notion suggests that the variation is some form of integration over the derivative. The following lemma will show that if f is continuously differentiable, this holds.

Lemma 2.10:
If $f \in C^1(\Omega)$, then the total variation is the integral over the absolute value of the gradient

$$\int_\Omega |Df| = \int_\Omega |\nabla f(x)| \, \mathrm{d}x. \tag{2.23}$$

Proof: In order to prove this, we first apply integration by parts to obtain the following expression.

$$\int_\Omega |Df| := \sup_{\substack{g \in C_0^1(\Omega, \mathbb{R}^d) \\ \|g\|_\infty \leq 1}} \int_\Omega f(x) \mathrm{div}(g)(x) \, \mathrm{d}x = - \sup_{\substack{g \in C_0^1(\Omega, \mathbb{R}^d) \\ \|g\|_\infty \leq 1}} \int_\Omega \sum_{i=1}^d \frac{\partial f}{\partial x_i} g(x)_i \, \mathrm{d}x \tag{2.24}$$

Since g is bounded by 1, we get

$$\int_\Omega |Df| \leq \int_\Omega |\nabla f(x)| \, \mathrm{d}x. \tag{2.25}$$

For the other inequality we choose as test function $g = \frac{\nabla f}{|\nabla f|}$. This is bounded by 1, and we obtain

$$\int_\Omega |Df| \geq \int_\Omega |\nabla f(x)| \, \mathrm{d}x, \tag{2.26}$$

which yields the assertion.

□

If we go back to the surface reconstruction problem for closed objects, our goal is to find a surface that is the boundary of an interior set. In the following, we derive a possibility to determine the reconstructed surface's size. If a boundary is smooth enough, we already know we can measure its size with the correct dimensional Hausdorff measure. If not, the concept of total variation allows us to define their size by looking at the total variation of the corresponding indicator function.

Definition 2.11:
Let $I \subset \Omega$. Then we define the **surface perimeter** by

$$\mathrm{per}_\Omega(I) := \int_\Omega |D\mathbb{1}_I|. \tag{2.27}$$

The following lemma shows why we can see the surface perimeter as a more general version of the boundary measure.

Lemma 2.12:
Suppose $I \subset \Omega$ has a C^2-boundary, then we can write the perimeter as

$$\mathrm{per}_\Omega(\mathcal{I}) = \mathcal{H}^{d-1}(\partial I). \tag{2.28}$$

Proof: Let $g \in C_0^1(\Omega)$ with $\|g\|_\infty \leq 1$. Then we can apply the Gauss-theorem and with $n : \partial I \to \mathbb{R}$ being the function that maps a vector to its outgoing normal, we obtain

$$\mathrm{per}_I(\Omega) = \sup_{\substack{g \in C_0^1(\Omega, \mathbb{R}^d) \\ \|g\|_\infty \leq 1}} \int_\Omega \mathbb{1}_I \mathrm{div}(g)(x)\, \mathrm{d}x = \sup_{\substack{g \in C_0^1(\Omega, \mathbb{R}^d) \\ \|g\|_\infty \leq 1}} \int_{\partial I} \langle g(x), n(x) \rangle \, \mathrm{d}\mathcal{H}^{d-1}(x). \tag{2.29}$$

Note that a scalar product attains its maximal value if the two vectors point in the same direction. This follows immediately from the connection between scalar products and the cosine function. Since both vectors are bounded by one, we get

$$\mathrm{per}_I(\Omega) = \int_{\partial I} \mathrm{d}\mathcal{H}^{d-1}(x). \tag{2.30}$$

□

For general sets, their perimeter does not need to be finite. If it is, we call the sets **Caccioppoli sets**. Now we want to look at the basic properties of these types of sets (chapter 3 in [Amb01]).

- If we scale up the domain that means we consider two domains $\Omega_1 \subset \Omega_2$, then

  $$\mathrm{per}_{\Omega_1}(I) \leq \mathrm{per}_{\Omega_2}(I). \tag{2.31}$$

  One can show, that equality holds, if $\overline{I}$ is compact in Ω_2.

- If we consider two different sets over the same space $I_1, I_2 \subset \Omega$, then we can estimate

  $$\mathrm{per}_\Omega(I_1 \cup I_2) \leq \mathrm{per}_\Omega(I_1) + \mathrm{per}_\Omega(I_2). \tag{2.32}$$

  In this case equality holds, if $d(I_1, I_2) > 0$.

- The map $I \mapsto \mathrm{per}_\Omega(I)$ is lower semicontinuous with respect to the $L^1_{loc}(\Omega)$ topology.

- If I is a set of measure 0, the perimeter of this set vanishes. From this one can conclude, that if the symmetric difference between I_1 and I_2, that are all point in Ω, that are either in I_1 or in I_2, has measure 0, then $\mathrm{per}_\Omega(I_1) = \mathrm{per}_\Omega(I_2)$.

- For any set $I \subset \Omega$, it holds

  $$\mathrm{per}_\Omega(I) = \mathrm{per}_\Omega\left(I^C\right). \tag{2.33}$$

Now let us get back to the surface reconstruction problem. There we considered a function

$u : \Omega \to \mathbb{R}$ describing a surface $\mathcal{S} = \{x \in \Omega \mid u(x) = 0\}$, and defined sets

$$\mathcal{I} = \{x \in \Omega \mid u(x) < 0\} \qquad \text{and} \qquad \mathcal{O} = \{x \in \Omega \mid u(x) > 0\}. \tag{2.34}$$

Now assume that $\mathcal{S}$ has measure 0. Then by the properties of the surface perimeter, it holds

$$\mathrm{per}_\Omega(\mathcal{I}) = \mathrm{per}_\Omega\left(\mathcal{I}^C\right) = \mathrm{per}_\Omega\left(\mathcal{O} \cup \mathcal{S}\right) = \mathrm{per}_\Omega\left(\mathcal{O}\right). \tag{2.35}$$

This shows that if we talk about the surface perimeter in this context, it does not matter if we consider the perimeter of the interior or the exterior. Therefore the sign of the function u is arbitrary.

A significant problem in determining the perimeter is that it is difficult to calculate analytically. Therefore, we are looking for functionals that Γ-converge to the perimeter function in the following.

Chapter 3

Variational formulations

The only thing missing for solving the surface reconstruction problem is finding an appropriate loss functional to train the neural network u_θ. We distinguish between two cases for loss functionals depending on the training data. The first one is that we already have a distance function, which might be discrete or continuous, and we want to convert it into a neural network. The second case is that we only have points sampled from the surface $\mathcal{S}$ as training data.
For the second case, if we have additional gradient information, one can get training data for a signed distance approach by sampling points as in (1.13). For an unsigned distance function approach, one could also use $d(x, \mathcal{P}) \approx d(x, \mathcal{S})$, but note that the distance function to the point cloud has far more discontinuities than to the surface since the cut locus is much larger. Now, having training data x_i from which we know, what the predicted output y_i should be, one way of defining a possible loss functional is

$$\mathcal{L}(u_\theta) = \sum_i |u_\theta(x_i) - y_i|. \tag{3.1}$$

Since we only care for good results near the surface and not so much if we are far away, one way to improve this loss was given in [PFS+19] by

$$\mathcal{L}(u_\theta) = \sum_i |\mathfrak{c}_\delta(u_\theta(x_i)) - \mathfrak{c}_\delta(y_i)| \tag{3.2}$$

where $\mathfrak{c}_\delta(x) := \min\{\delta, \max\{-\delta, x\}\}$ is called clamp function for some $\delta > 0$.
In addition, one can also add a surface error tolerance $\varepsilon > 0$, as described in [DZW+20], which gives an upper bound on the allowed error. The resulting loss functional is

$$\mathcal{L}(u_\theta) = \sum_i \max\{|\mathfrak{c}_\delta(u_\theta(x_i)) - \mathfrak{c}_\delta(y_i)| - \varepsilon, 0\}. \tag{3.3}$$

In practice, one would decrease ε during the training. In the beginning, the network learns a general and smooth surface and exposes more details over time. Apart from the clamp function, one can also use weights for the training data to ensure the learning focuses more around $\mathcal{S}$. One way of doing this was proposed in [CZ18] by using

$$\mathcal{L}(u_\theta) = \sum_i |u_\theta(x_i) - y_i| w_i \tag{3.4}$$

where w_i depends on the sampling density around point x_i. Another possible choice for the weights by [DNJ20] includes the distance of the point to the surface by choosing

$$w_i = \exp(-\beta d(x_i, \mathcal{S})). \tag{3.5}$$

Here, β is a parameter depending on the overall sampling distribution.
If we want to use the occupancy function approach, we are solving a binary classification problem. We, therefore, can use cross entropy classification as the loss functional, see [MON+19].

Now that we have handled the first case, we dedicate the rest of this chapter to the second, far more challenging one. We will start by finding a variational formulation for the phase field approach.

3.1 Phase transition problem

Before continuing on the surface reconstruction problem, we want to consider a physics-related problem: the phase transition problem. In this setting, we have a space Ω, and in this, we insert two different liquids. We want to find out how these liquids behave. Therefore we denote the density of the two liquids by a function $u\colon \Omega \to [-1, 1]$. For some position $x \in \Omega$, the case $u(x) = 1$ means that in this position, we have the first liquid, and $u(x) = -1$ means we have only the second liquid. The case $u(x) = 0$ means we have an equal mixing ratio of the two liquids. We will see that both fluids will arrange themselves so that the intersection among them has minimal area. That means we want almost every point within Ω to be either 1 or -1. To formulate this more mathematically, we need the following definition.

Definition 3.1:
A **double-well potential** W is a continuous function $W\colon \mathbb{R} \to [0, \infty)$, such that

(i) $W(x) = 0$ if and only if $x \in \{1, -1\}$.

(ii) There exist $L, R > 0$ such that $W(z) > L|z|$ for every $|z| \geq R$.

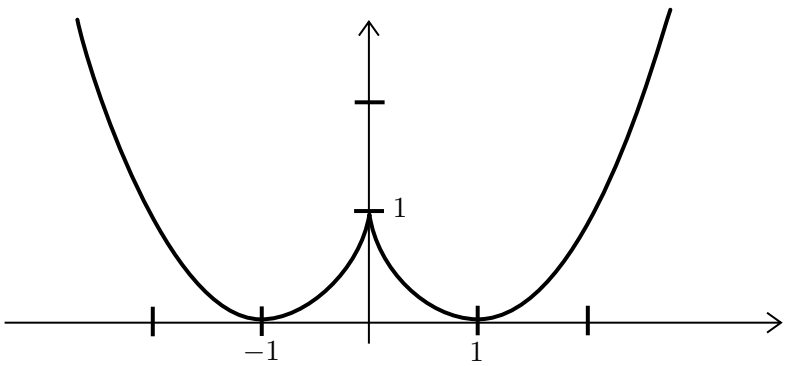


Figure 3.1: Graph of a double-well potential

Let W be a double-well potential. With the previous reasoning, we want $W(u)$ to be zero as much as possible. More mathematically speaking, we are looking for a solution for the following

minimization problem:

$$\begin{aligned} \min_{u \in L^1(\Omega,[-1,1])} \quad & \int_\Omega W(u(\xi))\,\mathrm{d}\xi \\ \text{s.t.:} \quad & \int_\Omega u(\xi)\,\mathrm{d}\xi = m \end{aligned} \tag{3.6}$$

The secondary condition comes from fixing a certain amount of the two liquids. In this, m is a given constant that measures the ratio of the liquids. For example, $m = 0$ means that both liquids' amounts should be the same.
One problem with the minimization problem stated above is that it does not minimize the area of interference of the liquids. Every decomposition of the space Ω, where u is 1 on the one part and -1 in the other part, minimizes this functional. So we want to change the minimizer to have only one minimum. We do so by adding a regularization term to it.

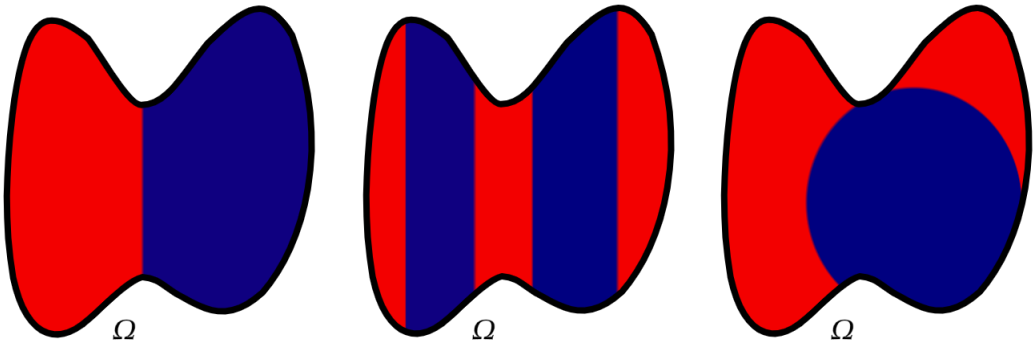


Figure 3.2: All these images are local minima of the phase transition energy, but we want the left one to be the only minimum since it has a minimal intersection area.

Adding a regularization term is similar to adding a regularizer term in machine learning. In our case, we decide to use a **Dirichlet regularizer**, which is defined as the integral over the absolute value of the gradient. In order to ensure existence, we minimize over the Sobolov space H^1. This means we end up with the following minimization problem that Modica and Mortola found in [MM77].

$$\begin{aligned} \min_{u \in H^1(\Omega,[-1,1])} \mathcal{M}_\varepsilon(u) &:= \int_\Omega \frac{1}{\varepsilon} W(u(\xi)) + \varepsilon \|\nabla u(\xi)\|^2\,\mathrm{d}\xi \\ \text{s.t.:} \quad & \int_\Omega u(\xi)\,\mathrm{d}\xi = m \end{aligned} \tag{3.7}$$

We now get a whole family of minimization problems that depend on a single parameter $\varepsilon > 0$. As ε decreases, the contribution of the double well potential increases while the one of the regularizer term decreases. Now we consider the limit case.

Theorem 3.2 (Modica-Mortola):
There exists a functional $\mathcal{M}\colon\ H^1(\Omega,[-1,1]) \to \mathbb{R}\cup\{\infty\}$, such that $\mathcal{M}_\varepsilon \xrightarrow{\Gamma} \mathcal{M}$, where

$$\mathcal{M}(u) := \begin{cases} C\cdot \mathrm{per}_\Omega(\mathcal{I}) & u\in BV(\Omega,\{1,-1\}) \\ \infty & \text{else} \end{cases}. \tag{3.8}$$

Here, $\mathcal{I}$ is again the interior defined by $\mathcal{I} := \{x\in\Omega \mid u(x)<0\}$. To show the connection from the Modica-Mortola functionals to the perimeter, define

$$H(t) := 2\int_{-1}^{t} \sqrt{W(s)}\,\mathrm{d}s. \tag{3.9}$$

Then for a function u fulfilling the total mass constrain the total variation of $H(u)$ is given by

$$\int_\Omega \|\nabla H(u(\xi))\|\,\mathrm{d}x = \int_\Omega 2\sqrt{W(u(\xi))}\|\nabla u(\xi)\|\,\mathrm{d}\xi. \tag{3.10}$$

Now since for every $a,b\in\mathbb{R}$ holds $a^2-2ab+b^2=(a-b)^2\geq 0$ and therefore $2ab\leq a^2+b^2$, the variation can be estimated by

$$\int_\Omega \|\nabla H(u(\xi))\|\,\mathrm{d}\xi \leq \int_\Omega \underbrace{\frac{1}{\varepsilon}W(u(\xi))}_{=a^2} + \underbrace{\varepsilon\|\nabla u(\xi)\|^2}_{=b^2}\,\mathrm{d}\xi = \mathcal{M}_\varepsilon(u). \tag{3.11}$$

If u_* is a proper occupancy function, that is $u_*(x)=1$ for $x\in\mathcal{O}$ and $u_*(x)=-1$ for $x\in\mathcal{I}$ and $\overline{\mathcal{I}\cup\mathcal{O}}=\Omega$, one can write $H(u_*)=H(1)\mathbb{1}_{\mathcal{O}}+H(-1)\mathbb{1}_{\mathcal{I}}$. Since $H(-1)=0$, we conclude

$$\int_\Omega H(1)\|\nabla\mathbb{1}_{\mathcal{O}}(\xi)\|\,\mathrm{d}\xi = H(1)\mathrm{per}_\Omega(\mathcal{I}). \tag{3.12}$$

In total, we have shown that

$$H(1)\mathrm{per}_\Omega(\mathcal{I}) \leq \mathcal{M}_\varepsilon(u) \tag{3.13}$$

with equality if

$$\int_\Omega 2\sqrt{W(u(\xi))}\|\nabla u(\xi)\|\,\mathrm{d}\xi \overset{!}{=} \int_\Omega \frac{1}{\varepsilon}W(u(\xi)) + \varepsilon\|\nabla u(\xi)\|^2\,\mathrm{d}\xi \tag{3.14}$$

holds. From this, one can conclude that the constant C from the Modica-Mortola theorem is given by $C=H(1)$.

3.2 Modica-Mortola for Surface reconstruction

This section and the next one summarize [Lip21]. We now want to utilize the variational approach we derived for the phase transition problem for the surface reconstruction problem. For this, we choose the occupancy function approach. The two liquids from before now correspond to the different occupancy values.
To derive the final loss functional we do two things. First, we transform the constrained min-

imization problem into an unconstrained one. Then we derive a signed distance function from the approximate occupancy function and use its properties for the loss function. So the final loss from [Lip21] creates a solution combining features of both these attempts. Nevertheless, before we do this, we want to state what a solution in this case is. Ideally, the minimizer u of the loss functional should fulfill the following properties.

(i) **Proper occupancy.** We want to solve the surface reconstruction problem using an occupancy function, so the image of u should be 1 or -1 almost everywhere. In other words we want $u \in BV(\Omega, \{\pm 1\})$.

(ii) **Zero reconstruction loss.** The solution should vanish along the point of the original point cloud, that is, $u(p) = 0$ for all $p \in \mathcal{P}$.

(iii) **Minimal perimeter.** If we demand the first two properties to hold, every arbitrary decomposition of Ω into an exterior and interior would be a solution. From all these decompositions, we want the one with minimal perimeter.

(iv) **Easy way of getting an SDF.** Given the approximative occupancy functions, we also want a way to construct a signed distance function. We do this to modify the final loss so that the properties of a distance function should hold.

If we apply the approach to the surface reconstruction problem, we can regard the first liquid as the interior of the surface and the second liquid as the exterior. Then, using the Modica-Mortola functional would already ensure the proper occupancy and minimal perimeter property. Since we do not have a mass constraint, we replace it with zero reconstruction loss by summing over the absolute values of the averaging integral over small balls around the points in the point cloud. We also do not treat this as a hard constraint but as a penalty term.

Modica-Mortola based

$$\min_{u \in H^1(\Omega,[-1,1])} \mathcal{F}_\varepsilon(u) := \int_\Omega \frac{1}{\varepsilon} W(u(\xi)) + \varepsilon \|\nabla u(\xi)\|^2 \, \mathrm{d}\xi + \frac{\eta}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \left| \fint_{B_\delta(p)} u(\xi) \, \mathrm{d}\xi \right|$$

Here, η is some positive constant, depending only on ε, that we specify later. By minimizing the functional, we want to enforce the averaging integrals to be 0 in these tiny balls of radius δ. Note that this is a stronger condition than just wanting the function to be zero in the underlying points from the point cloud because the integral becomes only zero if some parts of it cancel out. These parts come from one half lying in the interior and the other in the exterior; therefore, they have different signs. This should force the function to change its sign along the surface $\mathcal{S}$ and enforce the zero reconstruction property. Now similar to the Modica-Mortola theorem, we can prove the Γ-convergence of this new sequence of functionals. The only thing remaining is that we have to specify η. If ε decreases, the contribution of the double well potential, which works against the reconstruction loss, increases; therefore, η has to increase. The following theorem tells us at which speed precisely this has to happen.

Theorem 3.3 (Lipman):
(Theorem 2 in [Lip21]): If we choose $\eta = \eta(\varepsilon)$, in such a way, that $\lim_{\varepsilon\to 0}\eta = \infty$ and $\lim_{\varepsilon\to 0}\eta\sqrt{\varepsilon} = 0$, then $\mathcal{F}_\varepsilon \xrightarrow{\Gamma} \mathcal{F}_0$, where

$$\mathcal{F}_0(u) = \begin{cases} C \cdot \mathrm{per}_\Omega(\mathcal{I}), & u \in BV(\Omega, \{-1,1\}) \text{ and } \fint_{B_\delta(p)} u(\xi)\,\mathrm{d}m(\xi) = 0 \ \forall p \in \mathcal{P} \\ \infty, & \text{else} \end{cases}. \tag{3.15}$$

As in the Modica-Mortola case, we have $C = \int_{-1}^{1} W(u)\,\mathrm{d}\xi$. In the following we will use $\eta(\varepsilon) = \lambda\varepsilon^{-\frac{1}{3}}$. One problem with this theorem is that the limit functional is not quite what we want. For example, if we consider a point cloud sampled from a square, the zero reconstruction property will not hold in the corner points for the minimal perimeter solution for any δ. Moreover, this property does not hold in a perfect circle at any point.
Nevertheless, we will later see that we still get good approximations in practice.

3.3 SDFs and the Eikonal equation

In the previous chapter, we successfully introduced an approach for solving the surface reconstruction problem by learning an implicit representation of the underlying surface through an occupancy function. Now, we want to find a way of using the functionals $\mathcal{F}_\varepsilon$ and the corresponding minimizers from before to compute a signed distance function.
Therefore we have to adjust the one degree of freedom left that plays into the construction of $\mathcal{F}_\varepsilon$, which is the double-well potential. Any double-well potential is suitable for the previous theorem, but we need to state a function explicitly for the next computations. So from now on let

$$W(s) := s^2 - |2s| + 1. \tag{3.16}$$

We can see that this is a double-well potential and that it is the one shown in (3.1), which is non-differentiable at 0. Let u_ε be a minimizer of $\mathcal{F}_\varepsilon$, that we obtain using exactly this double-well potential. We already know that these converge to a suitable occupancy function, so now we want to apply a transformation map to these minimizers. Therefore we choose the so-called logarithm transform.

Definition 3.4:
Let $\varepsilon > 0$. For the minimiser u_ε of $\mathcal{F}_\varepsilon$ we define its **log-transform** by

$$w_\varepsilon(x) = \begin{cases} -\varepsilon\log(1 - u_\varepsilon(x)), & u_\varepsilon(x) \geq 0 \\ \varepsilon\log(1 + u_\varepsilon(x)), & u_\varepsilon(x) < 0 \end{cases}. \tag{3.17}$$

The log-transform is continuous and does not change the sign of the original function. The goal is to show that w_ε is an approximation of an SDF. To prove this, we will not directly use the variational problem of minimizing $\mathcal{F}_\varepsilon$ but show it through the partial differential equation that characterizes their minima.

Lemma 3.5:
(Theorem 3 in [Lip21]): Let $Q \subseteq \Omega - \bigcup_{p\in\mathcal{P}} B_\delta(p)$, with $u_\varepsilon(x) \neq 0$ for all $x \in Q$. Then on Q the function u_ε is smooth and fulfills the following partial differential equation

$$-\varepsilon^2 \Delta u_\varepsilon + u_\varepsilon - \operatorname{sign}(u_\varepsilon) = 0.$$

Proof: Note that since we are away from the original surface, W is differentiable with $W'(s) = 2s - 2\operatorname{sign}(s)$. We apply (2.14) with $F(\alpha, \beta, \gamma) = \frac{1}{\varepsilon} W(\beta) + \varepsilon\|\gamma\|^2$ and obtain

$$\begin{aligned} 0 &= \frac{\partial F}{\partial \beta} - \nabla \cdot \left(\frac{\partial F}{\partial \gamma}\right) \\ &= \frac{1}{\varepsilon}\left(2\beta - 2\operatorname{sign}(\beta)\right) - \varepsilon 2\nabla \cdot \gamma \\ \Leftrightarrow \quad &= -\varepsilon^2 \nabla \cdot \nabla u_\varepsilon + u_\varepsilon - \operatorname{sign}(u_\varepsilon). \end{aligned} \tag{3.18}$$

□

As ε tends to zero, the part containing the Laplacian vanishes, and only $u_\varepsilon = \operatorname{sgn}(u_\varepsilon)$ remains, which is precisely the occupancy property. Next, we insert the definition of the log-transform to find the characteristic equation for the transformed minimizers.

Theorem 3.6:
(Theorem 4 in [Lip21]): Let Q be the domain as in Lemma (3.5). Then on Q the log-transform w_ε satisfies

$$-\varepsilon \Delta w_\varepsilon + \operatorname{sign}(u_\varepsilon)\left(\|\nabla w_\varepsilon\|^2 - 1\right) = 0.$$

Proof: To shorten notation, we write $w = w_\varepsilon$ and $u = u_\varepsilon$ and the log-transform as $w = \mp \log(1 \mp u)$. Calculating the derivatives yields

$$\begin{aligned} \frac{\partial w}{\partial x_i} &= \varepsilon \frac{1}{1 \mp u} \frac{\partial u}{\partial x_i} \\ \frac{\partial^2 w}{\partial^2 x_i} &= \pm\varepsilon \frac{1}{(1 \mp u)^2} \left(\frac{\partial u}{\partial x_i}\right)^2 + \varepsilon \frac{1}{1 \mp u} \frac{\partial^2 u}{\partial^2 x_i}. \end{aligned} \tag{3.19}$$

From this, we conclude

$$\begin{aligned} -\varepsilon \Delta w_\varepsilon \pm \left(\|\nabla w_\varepsilon\|^2 - 1\right) &= \mp \frac{\varepsilon^2}{(1 \mp u)^2} \|\nabla u\|^2 - \frac{\varepsilon^2}{1 \mp u} \Delta u \pm \frac{\varepsilon}{(1 \mp u)^2} \mp 1 \\ &= -\frac{\varepsilon^2}{1 \mp u} \Delta u \mp 1 \\ &= \frac{1}{1 \mp u} \underbrace{\left(-\varepsilon^2 \Delta u + u \mp 1\right)}_{=0, \text{ because of Lemma (3.5)}}. \end{aligned} \tag{3.20}$$

□

Now we want to consider what happens if ε goes to zero. Again, the part with the Laplacian vanishes, and we obtain the following partial differential equation called the **Eikonal equation**

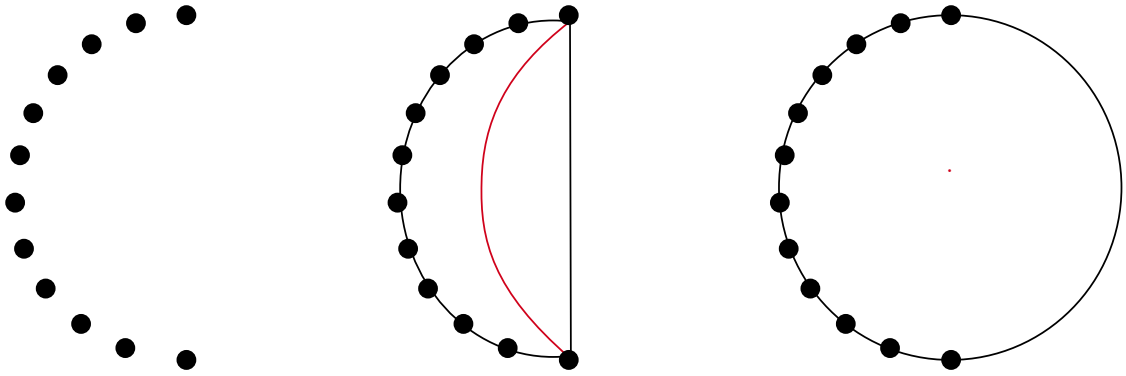
$$\|\nabla w\| = 1.$$

In Lemma (1.8), we have already seen that this is a necessary condition for a signed distance function. However, it is not a sufficient condition since, for example, a function that oscillates away from the surface in a zig-zag pattern also satisfies the Eikonal equation. Regardless, for this particular type of convergence, including the Laplacian, one can show that it indeed converges to a distance function, see Theorem 2.3 in [Var67].

Enforcing the Eikonal equation to hold already yields an approach to solve the surface reconstruction problem. This was shown in [GYH+20] and is called 'Implicit Geomemetric regularization'. In this paper, the authors propose a method to create smooth solutions via signed distance functions only by the Eikonal term and the zero reconstruction loss property. The loss function they use is the following.

$$\min_w \int_\Omega (\|\nabla w(\xi)\| - 1)^2 \, \mathrm{d}\xi + \frac{\eta}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} |w(p)| \tag{3.21}$$

However, there is no need for a minimizer of this functional to fulfill the minimal perimeter property. Since we enforce the Eikonal equation to hold everywhere, the solution to the minimization problem will be a function where this holds as much as possible. Therefore, the IGR loss minimizes the area of the cut locus, not the perimeter. In practice, we will see that this loss will always prefer the solution where the zero level set has as slight curvature as possible. For example, suppose we were to sample points from a half circle. In that case, the solution by the IGR loss will always be a full circle since, in this solution, the Eikonal equation holds everywhere except in a single point in the center.



Original Point Cloud | Minimal Perimeter | Minimal Cut locus

Figure 3.3: Minimizing perimeter (black) vs. minimizing cut locus (red) of points from a half circle

So let us get back to the Modica-Mortola-based loss. If we recall the assumptions of the last theorem, we needed to restrict our original domain to be away from the point of the point cloud so we could neglect the zero reconstruction loss. Therefore, we do not know if w fulfills the

Eikonal equation around these points or not. Hence, the idea is to modify the loss functional to enforce the Eikonal equation to hold around all points of the point cloud. Therefore, let $\mu > 0$ be a constant measuring the effect of the Eikonal term. Then we add the error of the Eikonal equation in the points to the loss. This final loss is called PHASE-loss.

PHASE-loss

$$\min_{u \in H^1(\Omega,[-1,1])} \mathcal{F}_\varepsilon(u) + \frac{\mu}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \|1 - \|\nabla w_\varepsilon(p)\|^2\|$$

Note that in the PHASE loss, w is the computed log-transform of u. Hence, the loss contains information on the approximate occupancy and signed distance functions. We get approximating functions for both by minimizing a single loss function. If we take the gradient of the log-transform in one of the points of the point cloud, i.e., where $u_\varepsilon(x) = 0$, we obtain

$$\nabla w_\varepsilon(x) = \nabla \left(\mp \varepsilon \log(1 \pm u_\varepsilon(x)) \right) = \mp \varepsilon \frac{\mp \nabla u_\varepsilon(x)}{1 \mp u_\varepsilon(x)} = \varepsilon \nabla u_\varepsilon(x). \tag{3.22}$$

That means we can calculate the gradient of the log transform without actually computing the log-transform.
If the original point cloud, in addition to the coordinates, also contains the normal vectors n in the points as features, we can rewrite the PHASE-loss as

$$\min_{u \in H^1(\Omega,[-1,1])} \mathcal{F}_\varepsilon(u) + \frac{\mu}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \|n(p) - \nabla w(p)\|. \tag{3.23}$$

3.4 Ambrosio-Tortorelli phase fields

The following part is based on [Cha00]. Using the occupancy approach, we successfully found an approach for solving the surface reconstruction problem in the previous chapter. Furthermore, we found a transformation, namely the log-transform, that delivers a signed distance function that also solves the initial problem. Note that we get both of these solutions by training a single neural network. From the signed distance function, we obtain a distance function by taking the absolute value. The last primary type of functions for implicit representations are indicator functions, for which we now want to derive a method to compute them.
The origin of this method does not lie in learning implicit functions, but this time from the area of image reconstruction. A black and white image is a matrix $G \in \mathbb{R}^{n,m}$ of grey level values $g_{ij} \in [0, 1]$. Now assume that the image we consider contains additive noise, which means it is the sum of an original perfect image U and a matrix N that contains random entries. For instance, one could assume that its entries are normally distributed around 0 with variance σ^2. Then we obtain

$$G = U + N. \tag{3.24}$$

Now we want to move from this discrete setting of matrices to a continuous one by regarding images as continuous functions $g, u, n : \Omega \to [0, 1]$. The space Ω can be considered the screen and is an open subset of $\mathbb{R}^2$, such as a rectangle. Again the evaluations of the function represent grey-level values of the images. We assume again that $g = u + n$ holds. For the normally distributed noise, we may assume

$$\int_\Omega n(x)\,\mathrm{d}x = 0 \qquad \text{and} \qquad \int_\Omega n(x)^2\,\mathrm{d}x = \sigma^2. \tag{3.25}$$

The problem of image reconstruction is how we can remove the noise from an image, or how do we get u from g without knowing n in advance?
There are many approaches to how one can handle this problem. The one we will discuss here is that we will write u as the solution to a minimization problem. The energy we want to minimize is called **weak Mumford-Shah energy** and was proposed in [MS89]. It is defined as follows

$$\mathcal{E}(u) := \int_\Omega |\nabla u(\xi)|^2\,\mathrm{d}\xi + \mathcal{H}^{N-1}(S_u) + \int_\Omega |u(\xi) - g(\xi)|^2\,\mathrm{d}\xi. \tag{3.26}$$

The first term means that the resulting image should be as smooth as possible. As S_u, we define the jump set of u. These are all points where u is not continuous. With $\mathcal{H}^{N-1}$ we denote the $N-1$ dimensional Hausdorff measure of a set. The last term ensures that our solution is close to the raw output data g.
The problem with this Mumford-Shah energy is that it requires knowledge of the jump set of u, which is numerically challenging to compute. That is why we consider a related family of functionals for $\varepsilon > 0$, that have been proposed by Ambrosio and Tortorelli in [AT90]

$$\mathcal{E}_\varepsilon(u, v) = \int_\Omega (v(\xi)^2 + k_\varepsilon)|\nabla u(\xi)|^2 + \varepsilon|\nabla v(\xi)|^2 + \frac{(1 - v(\xi))^2}{4\varepsilon} + |u(\xi) - g(\xi)|^2\,\mathrm{d}\xi. \tag{3.27}$$

Here, k_ε is a constant, depending only on ε. We now minimize over two argument functions in this new family of functionals. The idea is that as ε goes to 0, v should approximate the indicator function of the complement of the jump sets of u. That means v should be almost everywhere equal to 1, but in the area where u makes a jump, and there v should drop down to 0. Therefore, the Ambrosio-Tortorelli energy is an approximate weak Mumford-Shah energy, and we can prove Γ-convergence of these types of functionals.

Theorem 3.7:
(Theorem 4 in [Cha00]): If k_ε goes faster to zero, than ε does, then $\mathcal{E}_\varepsilon \xrightarrow{\Gamma} \mathcal{E}_0$, where

$$\mathcal{E}_0(u, v) := \begin{cases} \mathcal{E}(u) & \text{if } u \in GSBV(\Omega),\ v(x) = 1 \text{ a.e. in } \Omega \\ \infty & \text{else} \end{cases}. \tag{3.28}$$

Here $GSBV$ is the space of functions with bounded variation that does not include fractals, like the devil's staircase. Now we want to get back to the surface reconstruction problem. We know that the jump set of an occupancy function equals its zero level set and, therefore, the underlying surface. Therefore, we want to use the v in the Ambrosio-Tortorelli energy for the implicit representation. Since we no longer have a raw image as information, we remove the

similarity condition $\|u - g\|$. Now, we can observe that the Ambrosio-Tortorelli energy gets minimized if we choose u to be any constant function, and therefore the functional only depends on v.

Now we have to think about how we can enforce the zero reconstruction property to hold, which means $v(p) = 0$ for all $p \in \mathcal{P}$. We can not take the averaging integral over a small ball around the points since there is no longer a change of sign around them. The easiest way is by adding the sum of the absolute values of the function in the points of the point cloud since we expect the function to be zero here

$$\frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} |v(p)| \,.$$

For the final loss, this yields

Ambrosio-Tortorelli-based

$$\min_v \int_\Omega \varepsilon |\nabla v(\xi)|^2 + \frac{(1 - v(\xi))^2}{4\varepsilon} \, \mathrm{d}\xi + \frac{\eta}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} |v(p)|$$

This looks similar if we compare this to the Modica-Mortola-based approach. There, we minimize a double-well potential, forcing every point to be classified as either an interior point or an exterior point. These two classes correspond to the two global minima of the double well potential. On the other hand, in the Ambrosio-Tortorelli-based approach, we assign almost every point to only one class, which contains the points not directly on the surface. Therefore, this is nearly the same as the functional we get by replacing the double-well potential in the Modica-Mortola-based approach with a function with only one global minimum in $+1$, namely $U(s) := (1 - s)^2$. Compared to the double-well potential, this is convex and everywhere differentiable.

In summary, we constructed a new loss functional, whose minimizers are mostly 1, but in the vicinity of the point cloud, they continuously bounce to 0.

The signed distance and occupancy functions separate the space in an interior and exterior. The classification is visible in the sign of the function. For the Ambrosio-Tortorelli-based approach, such a distinction does not exist. Therefore, these functions can reconstruct shapes that are not closed, like open curves. However, there is one big problem with this approach. Since a single point can be considered a constant open curve, the reconstructed surface does not need to be connected, even if the original surface from which the points originate is smooth. Moreover, as ε tends to zero, the Ambrosio-Tortorelli phase field functions will not converge to the minimal perimeter solution. However, later we will see that if we do not choose ε too small, we will still get good results.

We do not know if one can show Γ-convergence if we increase the point cloud's density, as ε tends to zero and leave this for future work.

3.5 Optimal Profiles

Summing up, we found ways to approach the surface reconstruction problem using all the initially discussed types of functions. Before we continue with their numerical implementation, we want to consider a case, where we can solve the variational problems of the last chapters analytically. By minimizing $\mathcal{F}_\varepsilon$, we get an approximation u_ε of an occupancy function. Applying the log-transform gives an approximation w_ε of a signed distance function. As ε tends to zero, the zero-level set of all these functions converges to the minimal perimeter solution for the surface reconstruction. If we use a single-well potential instead of a double-well potential, we get an Ambrosio-Tortorelli-like function v_ε, which approximates the indicator function. To visualize their differences, we consider a simple 1D example, where the point cloud consists of a single point $\mathcal{P} = \{0\}$ and look at the different solution functions. These are called the **optimal profiles**.

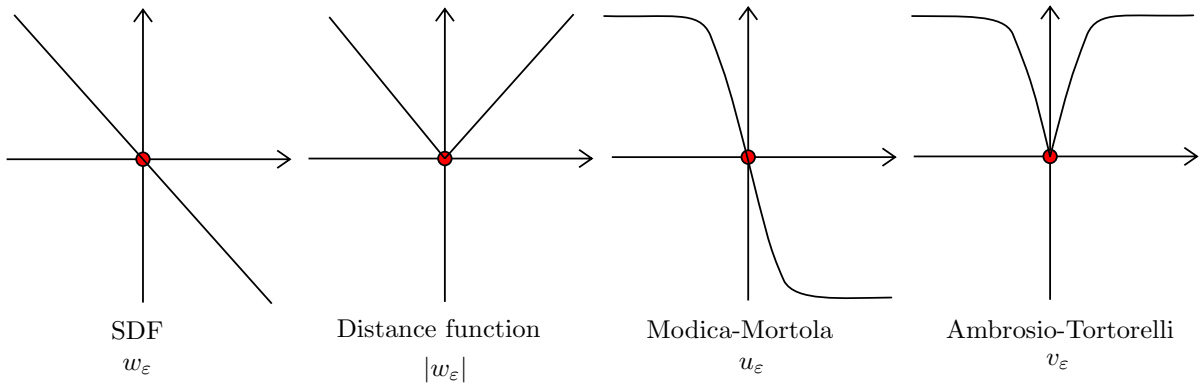


Figure 3.4: A 2D-example for the setting of surface reconstruction

If we treat the zero reconstruction loss property as a necessary constraint, it is possible to compute the minimizers directly. Therefore, we minimize the Modica-Mortola and Ambrosio-Tortorelli terms under the constraint $u(0) = 0$. We start with the SDF function and the Modica-Mortola functional $\mathcal{M}_\varepsilon$. Using the substitution $t = \frac{x}{\varepsilon}$ and similar computations as in (3.11), we deduce

$$\begin{aligned}
\mathcal{M}_\varepsilon(u) &= \int \frac{1}{\varepsilon} W(u(x)) + \varepsilon |u'(x)|^2 \,\mathrm{d}x \\
&= \int \left(\frac{1}{\varepsilon} W(u(t)) + \frac{\varepsilon |u'(t)|^2}{\varepsilon^2} \right) \varepsilon \,\mathrm{d}t \\
&= \int W(u(t)) + |u'(t)|^2 \,\mathrm{d}t \\
&\geq \int 2\sqrt{W(u(t))} |u'(t)| \,\mathrm{d}t \\
&= 2 \int_{-1}^{1} W(s) \,\mathrm{d}s \\
&= C.
\end{aligned} \tag{3.29}$$

We have shown that $\mathcal{M}_\varepsilon$ is bounded from below by C and that this bound is attained for

$$W(u(t)) + |u'(t)| = 2\sqrt{W(u(t))}|u'(t)|. \tag{3.30}$$

It is easy to see, that if $u'(t) = \sqrt{W(u(t))}$ holds, than this constraint is fulfilled. Therefore, for computing the optimal profile, it is sufficient to solve the ordinary differential equation

$$\begin{aligned} u'(t) &= \sqrt{W(u(t))}, \\ u(0) &= 0. \end{aligned} \tag{3.31}$$

For the double well potential of the PHASE-loss $W(s) = s^2 - 2|s| + 1$, we get

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \sqrt{u^2 - 2|u| + 1}. \tag{3.32}$$

By separation of variables, we get

$$\int \frac{1}{\sqrt{u^2 - 2|u| + 1}} \mathrm{d}u = \int 1 \,\mathrm{d}t = t + C. \tag{3.33}$$

Since $\frac{\mathrm{d}|u|}{\mathrm{d}u} = \operatorname{sgn}(u)$ we substitute $\tilde{u} = |u|$ and get

$$\int \frac{1}{\sqrt{u^2 - 2|u| + 1}} \mathrm{d}u = \operatorname{sgn}(u) \int \frac{1}{\sqrt{(1-\tilde{u})^2}} \mathrm{d}u = \operatorname{sgn}(u) \log(1 - \tilde{u}). \tag{3.34}$$

Now assume $u \geq 0$, then

$$\begin{aligned} \log(1 - u) &= t + C \\ u &= 1 - e^{t+C}. \end{aligned} \tag{3.35}$$

If $u \leq 0$ we get

$$\begin{aligned} -\log(1 + u) &= t + C \\ u &= e^{-(t+C)} - 1. \end{aligned} \tag{3.36}$$

From $u(0) = 0$ we conclude that $C = 0$. Note that $u \geq 0$ if and only if $t \leq 0$ and therefore we get the minimizer

$$u(t) = \begin{cases} 1 - e^t & t \leq 0 \\ e^{-t} - 1 & t \geq 0. \end{cases} \tag{3.37}$$

And by resubstituting $t = \frac{x}{\varepsilon}$ we get the optimal profile

$$u_\varepsilon(x) = \begin{cases} 1 - \exp\left(\frac{x}{\varepsilon}\right) & x \leq 0 \\ \exp\left(-\frac{x}{\varepsilon}\right) - 1 & x \geq 0. \end{cases} \tag{3.38}$$

Note that applying the log-transform to these minimizers gives the negative identity

$$w_\varepsilon(x) = \mp\varepsilon \log(1 \mp u_\varepsilon(x)) = -x. \tag{3.39}$$

This is indeed the signed distance to the origin.
The specific choice of the double-well influences the optimal profile. For example if one chooses $\tilde{W}(s) = \left(1 - s^2\right)^2$ the corresponding optimal profile would be $\tilde{u}_\varepsilon = \tanh\left(\frac{x}{\varepsilon}\right)$, see [Bra02].

For the Ambrosio-Tortorelli functional $F = \frac{1}{4\varepsilon}(1-v)^2 + \varepsilon|v'|^2$ a similar computation yields

$$v_\varepsilon(x) = 1 - \exp\left(-\frac{|x|}{2\varepsilon}\right). \tag{3.40}$$

To verify this formula, we check whether the Euler-Lagrange equation (2.14) holds, see Lemma 2.1 in [Fra12]. First, for $v_\varepsilon \geq 0$

$$\begin{aligned}
\frac{\partial F}{\partial v_\varepsilon} - \nabla \frac{\partial F}{\partial v'_\varepsilon} &= -\frac{1}{4\varepsilon} 2(1 - v_\varepsilon) - 2\varepsilon v''_\varepsilon \\
&= -\frac{1}{2\varepsilon}\left(1 - \left(1 - \exp\left(-\frac{|x|}{2\varepsilon}\right)\right)\right) + 2\varepsilon \frac{1}{4\varepsilon^2} \exp\left(-\frac{|x|}{2\varepsilon}\right) \\
&= 0.
\end{aligned} \tag{3.41}$$

The case $v_\varepsilon \leq 0$ goes analogous.

Chapter 4

Improving the network

Before we continue with the numerical experiments, there are some adjustments we need to make to the neural network u_θ. We have already seen that neural networks can approximate any function. To ensure that the network approaches the solution we want, we need to make a few improvements compared to the standard MLP. We already know that we can write the deep learning problem as a minimization problem by

$$\min_\theta \mathcal{L}\big((u_\theta(\vec{x}_i))_{i=1}^N, (y_i)_{i=1}^N \big). \tag{4.1}$$

Since the loss functionals we consider contain integrals, which are impossible to solve analytically, we approximate them using Monte Carlo integration. This means we sample points $\{x_i\}_{i=0}^N$ uniform in the domain Ω and approximate

$$⨍_\Omega f(u_\theta(\xi))\,\mathrm{d}\xi \approx \frac{1}{N}\sum_{i=0}^{N} f(u_\theta(x_i)). \tag{4.2}$$

In this case, one can show that the approximation converges with order $\mathcal{O}(N^{-\frac{1}{2}})$, see [LP14]. We adjust the weights using an optimization algorithm. The one we use is called Adam optimizer and belongs to the gradient descent algorithms class, see [KB14]. Optimizing a function using these algorithms always works after the same scheme: We start at a randomly generated starting point. In our case, a "point" is a vector representing the different weights in the network. After evaluating the loss functional $\mathcal{L}$, we need a way to adjust the neural network weights. This is done using **backpropagation**. It means we calculate the gradient in the current error of the loss functional with respect to the weights. This indicates how much to the error the weights $\tilde{\theta}$ of the last hidden layer contribute. We take adjustments

$$\tilde{\theta} \leftarrow \tilde{\theta} - h\frac{\partial \mathcal{L}}{\partial \tilde{\theta}}.$$

Here, h is the learning rate. A special feature of the Adam optimizer is that it can store different learning rates for individual weights. We then use the chain rule for differentiation iterative to update the weights in the previous layers of the network. Then we take the step and repeat the process until we are sufficiently close to the minimum. At this point, the gradient equals zero. However, since the loss functionals we consider are not convex, multiple global or local minima

may exist. Therefore, by choosing the right starting point, we can guarantee to converge to the right one, which is what we will do in the next section.

4.1 Geometric initialization

The first thing we want to take care of is the convergence of the network's weight. For this, we focus only on the Modica-Mortola-based approach.
To ensure that we converge against the right minimum, we must find initial weights close to the correct solution. Thereby, we also improve the convergence speed.
To do this, we consider a new loss function for the surface reconstruction problem from [AL19]. In this paper, they learn a signed distance function as the minimizer of the following functional.

$$\min_u \int_\Omega \left| |u(\xi)| - d\left(\xi, \mathcal{P}\right)\right| \, \mathrm{d}\xi \tag{4.3}$$

The idea here is that $|u|$ approximates the unsigned distance to the underlying point cloud $\mathcal{P}$ in terms of the metric d. The signed distance function would be a solution by choosing the metric to be the standard Euclidean distance. To improve the loss function, one can also compare the network function's unsigned gradients with the distance functions gradient, see [AL20].
However, the problem is that there are multiple local minima to this problem. On the one hand, we have the desired signed distance function as a solution, and on the other hand, an unsigned distance function, where u does not change the sign along the original points. In order to prevent the wrong convergence, the authors use a **geometric initialization**, which is a proper initialization of the network parameters. It is defined as follows: First, we initialize all weights that are associated with hidden layers and do not come from bias terms, randomly from a normal distribution with mean value 0 and variance $\sqrt{\frac{2}{d_i}}$. Here d_i is the number of neurons in the next layer. Then we set all the corresponding bias parameters to zero. In the final output layer, we set the weights to $\sqrt{\frac{\pi}{d}}$, with d being the output dimension of the neural network and the final bias term to $-r$ for some $r > 0$. If we do this, we obtain the following theorem.

Theorem 4.1:
(Theorem 1 in [AL19]): Suppose we sample a point cloud from a sphere with radius r. Then, using the geometric initialization described above, the neuronal network will converge to approximate the signed distance function solution.

This proof might only yield convergence for sphere-shaped objects, while in reality, we get good results on all kinds of data input. Now we want to get back to the PHASE loss. Even though this loss differs from the SAL loss, the geometric initialization still provides a reasonable starting point for the gradient descent.

4.2 Skipping layers

Now we want to take care of another problem. The following part is based on [Ada20]. We want our network also to handle larger point clouds and more complicated objects. For this to work, we need to increase the number of neurons and layers in our network to expect

good results. However, this extension comes with a price called **vanishing gradient problem**. It arises because we apply the chain rule for backpropagation. For example, if we consider a single weight w in the i-th layer of a neuronal network with a total of l layers, we compute the gradient of the loss functional with respect to w as the product of $l - i$ gradients with respect to the previous weights. In practice, most of the gradients that appear and the learning rate will have absolute values much smaller than 1. So, if we have many layers in the network, the gradient in the early layers will be close to zero and vanish. However, this causes the parameters of these early layers to not update during the network training. Therefore, we can conclude that just increasing the layer sizes will not yield better learning.

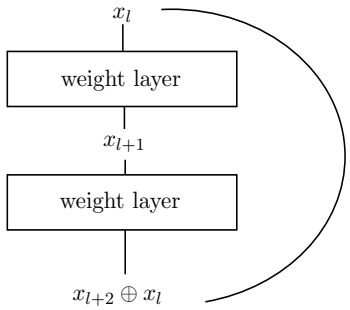


Figure 4.1: ResNet

One way of dealing with this problem is by not using a simple, fully connected network but a deep residual learning framework, as proposed in [HZRS16]. The idea is to build alternative paths for the gradients to be directly connected to the early layers without passing through all layers. The idea is to add a connection to the rear layers resembling the identity from the input layer. We do this by concatenating the feature vector with the original input. These connections are called **skipping layers**. They allow the backpropagation of the error into the early layers without passing through the whole network, prohibiting the vanishing gradient problem.

There is another view on skipping layers that is not important for this work but, at this point, worth noticing. It was first introduced in [Wei17]. Therefore, consider a fully connected network, where every layer is given as a function $\mathscr{F}$. Let x_l be the output of layer l such that $x_{l+1} = \mathscr{F}(x_l)$. If we add a single skipping layer that skips only layer l, then the output of this layer is given by

$$x_{l+1} = x_l + \mathscr{F}(x_l).$$

This is reminiscent of a single step in the implicit Euler method, which is an iterative method for solving partial differential equations. The corresponding equation is given by $\frac{\mathrm{d}x}{\mathrm{d}t} = \mathscr{F}(x)$ and we choose a constant step size of 1 for the discretisation. This shows a connection between discrete dynamic systems and deep neural networks with skipping layers.

4.3 High frequency learning

There is a third and last adjustment left to improve the network. Therefore, we have to consider again the way networks learn. Let us consider a deep neural network that is large enough and there is sufficient training data available so that the network can give a good approximation. We still end up getting only a very rough reconstruction of the surface.

This is because the optimization algorithm also only gives an approximation to the solution. In the case of MLPs, where we use a stochastic gradient descent algorithm and tend the learning rate to zero, one can show convergence of this method. In this case, it is possible to give an explicit bound for convergence speed. It has been shown, for example, in [BJKK19, RBA$^+$18], that the convergence speed depends on the frequencies of the target function. Low frequencies

converge faster and therefore are learned earlier than high frequencies. This is why early stopping in neural network training can prevent the model from overfitting. Nevertheless, in the case of surface reconstruction, high frequencies of the target function correspond to high levels of details of the object. As opposed to many of the typical application areas of neural networks, we want to enforce overfitting in the case of learning geometric data to a certain degree.
In order to do this, we want to shift the frequencies by building periodic functions into the network architecture. The easiest way of doing this is by using periodic activation functions. One example by [SMB+20] called SIREN uses MLPs with sine functions as activations in every neuron. By doing this, we still have a universal approximation theorem for these types of networks, see [Can99].

A different approach is to embed the points into a higher dimensional space before we pass them into the network. Such a mapping is called **feature embedding**. Then, the high dimensional data is used as the input for the neural network. The goal is to create an embedding that allows us to control the bandwidth of the learned function. The embedding we choose is the **Fourier feature mapping** by [TSM+20], which is a class of functions that uses trigonometrical functions for the embedding. For an input point $x \in \mathbb{R}^d$ and vectors $b_m \in \mathbb{R}^d$ for $1 \leq m \leq M$, the Fourier feature map is defined as a function

$$\gamma(x) := \Big(\cos(2\pi b_1^T x), \quad \sin(2\pi b_1^T x), \quad \dots \quad \cos(2\pi b_M^T x), \quad \sin(2\pi b_M^T x)\Big)^T. \tag{4.4}$$

The Fourier feature embedding maps inputs to the surface of a higher dimensional hypersphere in dimension $2M$. The idea behind Fourier features is that the b_m corresponds to Fourier basis frequencies and represents the types of frequencies the network should learn. If we do not have any prior information on the frequencies of the target function, we initialize these vectors randomly. Experiments have shown that the exact sampling distribution is much less important than the standard deviation of the distribution. To shorten notation, we can write $\gamma(x) = \Big(\cos(2\pi Bx) \quad \sin(2\pi Bx)\Big)^T$, where the trigonometric functions are applied element-wise and $B \in \mathbb{R}^{M \times d}$. The entries of B are samples from a normal distribution $\mathcal{N}(0, \sigma^2)$, where σ is the standard deviation that depends on the underlying problem. Changing the value of σ makes it possible to change the convergence behavior of the network significantly. If σ is too low, only low frequencies are learned, and details can not be represented accurately, causing underfitting. If σ is too high, no low frequencies are learned and therefore cause overfitting. So, the standard deviation for selecting the matrix entries does the same as the regularization parameter. Fourier features can be seen as a one-layer SIREN network, with the difference being that in the SIREN network, B is learnable.

Chapter 5

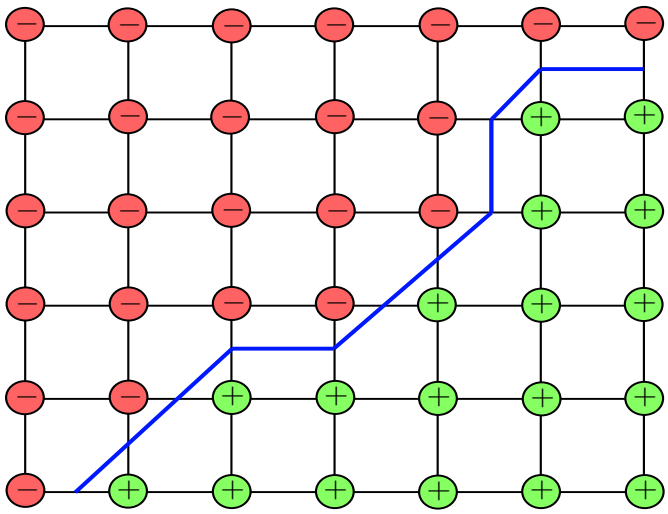
Numerical implementations

5.1 Rendering

In order to visualize our results, we would like to produce images of the reconstructed surfaces. That means that given the implicit function, we want to be able to produce a picture that shows the zero-level set of the function. This process is called **rendering**. Since all images are constrained to a certain resolution, such visualization will always yield a loss of information. In the following, we will see examples of how rendering works. The easiest way to render an object is if that object is given explicitly. Therefore, we will first explore a way to convert a level set into an explicit representation.

5.1.1 Marching Cubes

The most common algorithm for extracting level sets is called **Marching Cubes**, which was initially designed in [LC87] to handle medical data from computed tomography or magnetic resonance. This algorithm produces a mesh of the level set from discrete data, which standard rendering algorithms can handle.
First, we put a rectilinear grid in the interior of our input domain and evaluate the target function on all the vertices to get a discrete representation of the target function. Now the algorithm works by the divide and conquer principle, which means we process every grid cell separately. We start by assigning a label to each vertex in every grid cell. This label is "+" if the function value is above the value of which we want to extract the isosurface and "-" otherwise. These labels determine the behavior of the surface within each cell. For example, if all labels are the same, nothing has to be done. However, if at least one label differs from the other, then the level set must cross the cell. Since there are 2^d vertices in each cell and only two possible labels, we get 2^{2^d} possible ways the surface can intersect the cell. Using the fact that some of these combinations are the same by rotation or inverting the labels, one can reduce this number to only four unique cases for $d = 2$ and 14 cases for $d = 3$. We can store all of these cases in a look-up table that tells which edges intersect. Then we decide for every cell which of the cases is present and interpolate the surface intersection linearly at these edges. From these interpolations, we can construct a part of the mesh for this cube. Additionally, we can also compute the triangle's normal vector and store it.



Algorithm 1 Marching Cubes

```
Create look-up table
for every vertex v do
    if f(v) ≥ 0 then
        assign v with label "+"
    else
        assign v with label "-"
    end if
end for
for every cell c do
    determine state of c from look-up table
    Compute linear interpolation
    Create mesh
end for
```

Figure 5.1: 2D Marching Cubes Example

After doing this, one can combine all the parts for every cell and obtain a mesh structure of the level set and the corresponding surface normals.

5.1.2 Ray tracing

We can render images of 3D meshes using a **ray tracing** algorithm. Ray tracing for computer graphics was first accomplished in [App68], yet the idea has already existed since the 16th century. First, we need to choose a viewing point s. Then we embed a 2D image plane into the 3D space between the object and the viewing point. For each pixel p in the image plane, we need to create a ray r. A ray is a straight line from the viewing point to the specific pixel. It can be written as

$$r(t) = s + t(p - s) \qquad t \in \mathbb{R},\ t \geq 0 \tag{5.1}$$

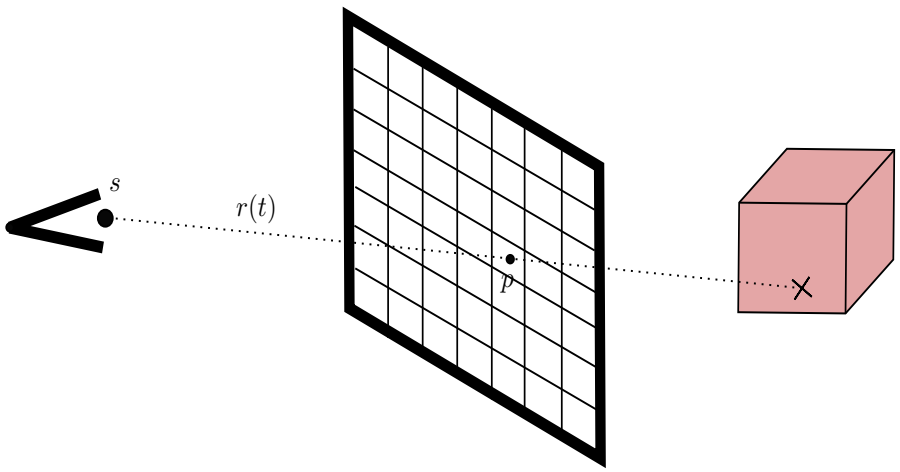


Figure 5.2: Ray tracing

We can compute the possible intersection with the triangular mesh for each of these rays by solving a linear system of equations. If at least one intersection exists, we choose the one closest

to the viewing point, i.e., the one with the smallest t value. Then we add a predefined color value of the object to the pixel on the image plane. This is only an elementary version of raytracing. If we want to use some shading in addition, we multiply the color value with a term that depends on the angle between the incoming ray and the surface's normal vector. This makes the final image look more realistic.

5.1.3 Sphere tracing

For all our numerical experiments, we will use the marching cubes pipeline above for the visualization because it is swift and produces good results. However, it is not ideal, so we want to mention the reasons and alternative rendering schemes. There are mainly two problems with Marching Cubes. One problem is that the target function must cross that particular value to plot a level set. This is, for example, a problem when we want to extract the zero-level set of an Ambrosio-Tortorelli phase field. These functions reach the value 0 but do not cross; therefore, Marching Cubes does not work. However, one can work around the problem by not extracting the 0 level set but the c level set for $0 < c \ll 1$. The other problem is that we do not use the properties of the implicit function since we convert it in the first step into a mesh.
One algorithm, that can directly render images from implicit functions is **sphere tracing** [Har96]. Therefore, we assume that a signed distance function w is given. We fix a viewing point again and consider rays through an image plane as in (5.1). Now let $F := w \circ r$. We need to find points where the ray intersects the zero level set, which means we want to find out if there is some point in time t with $F(t) = 0$.
If F is differentiable, one could use algorithms such as Newton's method, but there is an easier way, where we only need that F is Lipschitz continuous. Therefore, we define a sequence $\{t_i\}$ recursively by

$$t_{i+1} = t_i + F(t_i). \tag{5.2}$$

Since $F(t_i)$ is the minimal distance to the surface, we can make a step of this size and guarantee that we will not intersect the surface. The sequence converges if and only if there is an intersection point. In this case, $(F(t_i))$ converges linearly to the first position, where the ray hits the 0-level set. In practice, we compute new sequence elements until $F(t_i) < \varepsilon$ or a certain number of steps has been made.
The sphere algorithm also works with proper unsigned distance functions v, but since we only consider approximation through neural networks, we might run into some trouble here. In this case, we only move forward along the ray since $\tilde{F} = v \circ r \geq 0$. Therefore we might overshoot a possible zero crossing of $\tilde{F}$. Two modifications for the sphere algorithm that counteract this problem were proposed in [CMPM20]. The first one is that we reduce the step size by setting

$$t_{i+1} = t_i + \alpha \tilde{F}(t_i) \qquad \text{for } 0 \leq i \leq N \tag{5.3}$$

for some $\alpha \in (0, 1]$. We repeat this until sphere tracing terminates at step N. If we still overshoot, we need to move back along the ray but now use a different update rule. From the Taylors

expansion

$$0 = \tilde{F}(t) = v(s + t(p - s)) \approx v(s) + t(p - s)^T \nabla v(s) \tag{5.4}$$

we conclude the second update rule

$$t_{i+1} = t_i + \beta \frac{-v(p)}{(p - s)^T \nabla v(s)} \qquad \text{for } i \geq N \tag{5.5}$$

with stepsize reduced by some factor $\beta \in (0, 1]$. We iterate this again until $\tilde{F}(t_i) < \tilde{\varepsilon} < \varepsilon$.

One alternative is to move the ray along an interval $t \in [a, b]$. Then in every step we check if $F([a, b])$ contains 0. If it does, we decrease b. If not, we take the step. Bounding the image of an interval for a neural network can be done efficiently using Range Analysis, see [SJ22].

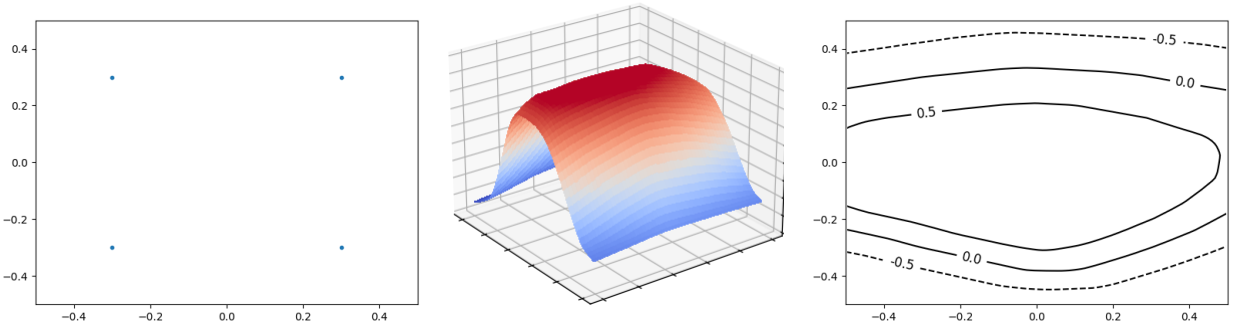
5.2 Modica-Mortola-based surface reconstruction

We want to look at the numerical experiments of the PHASE loss. We do this by using the `pytorch` environment of `python`, the Facebook Research group developed, see [PGM+19]. The network for the initial experiments is a multi-layer perceptron with three hidden layers of each 128 neurons with softmax activation functions. A softmax function $\sigma(x) = \frac{1}{\beta} \log(1 + \exp(\beta x))$ is a smooth approximation of a ReLU, that converges to it as $\beta \to \infty$. We use them to ensure that our network is differentiable so that we can compute its gradients with respect to the input value. In the output layer, we use tanh as an activation function, ensuring the network range is in $[-1, 1]$. The weights are randomly initialized and optimized using Adam optimizer. We start by using a learning rate of 0.01 at let it decrease by a factor of 0.1 automatically if, after 1500 training epochs, there was no improvement in the total loss. We start by using the PHASE-loss functional for the training. Therefore, we need to implement

$$\min_\theta \int_\Omega \frac{1}{\varepsilon} W(u_\theta(\xi)) + \varepsilon \|\nabla u_\theta(\xi)\|^2 \, \mathrm{d}\xi + \frac{\lambda \varepsilon^{-\frac{1}{3}}}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \left| \fint_{B_\delta(p)} u_\theta(\xi) \, \mathrm{d}\xi \right| + \frac{\mu}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} |1 - \|\nabla w_\theta(p)\||^2. \tag{5.6}$$

We choose $\Omega = [0.5, -0.5]^d$ for $d = 2, 3$ and at first scale our input point cloud $\mathcal{P}$ in such a way, that $\mathcal{P} \subset [-0.3, 0.3]^d$ so that we stay away from the boundary. We estimate the integrals of PHASE-loss by using Monte-Carlo integration with 200 samples for the integral over Ω and 50 sample per integral over $B_\delta(p)$. In practice, we can see that the number of integral samples can be arbitrarily low if the number of training steps is sufficiently high since new points for the Monte Carlo evaluation are drawn in every training iteration. Instead of taking values in a ball with radius δ, we sample points normally distributed with expectancy p and variance $\sigma = 0.001$ since this is much faster. The double well potential is the one in (3.16). For the other parameters, we start by choosing $\varepsilon = 0.01$, $\lambda = 14$ and $\mu = 1$. We will later see the effects of increasing/decreasing the different parameters. The network function's gradients are automatically computed using PyTorchs auto differentiation.
Now we want to take a look at some experimental results. For extracting isosurfaces, we use the

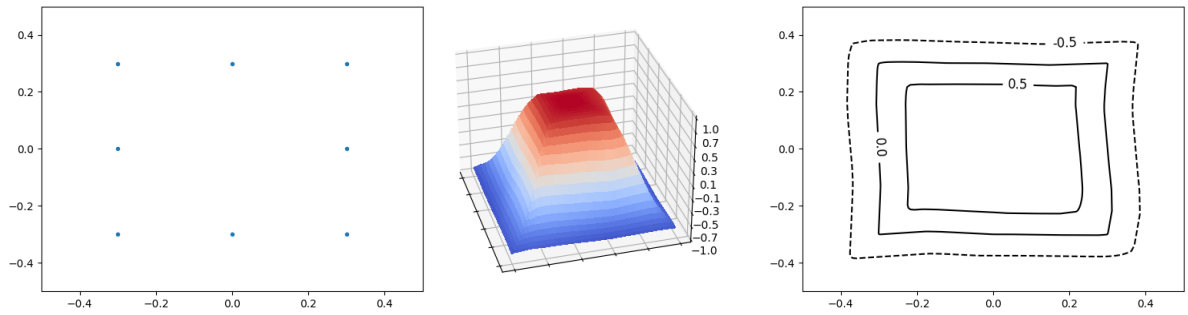
Marching Cubes algorithm with a resolution of 100^2. To start, we sample points from a square.



(a) Original point cloud (b) 3D graph (c) Contour plot

Figure 5.3: Reconstructing a square from four points

The reconstruction is not a square since the distance to the boundary is shorter for every point than for the next point. Therefore the perimeter gets minimized by connecting every two points with two straight lines. In order to obtain a square, we need to either scale the point cloud down or increase the number of sampled points.



(a) Original point cloud (b) 3D graph (c) Contour plot

Figure 5.4: Reconstructing a square from 8 points

As hoped, we can see a successful reconstruction of the square by increasing the number of points. Even though the lines are not perfectly straight, we can recognize the shape of the square. However, the plot we can see in (5.4) is not the only result we get while training the network. Sometimes, the network will converge to the following function, which is mostly 1 and has spikes around the points of the point cloud that ensure that the integrals over the δ-balls get minimized. By the convergence behavior of the network parameters, in the end, we can see that this also is a local minimum.

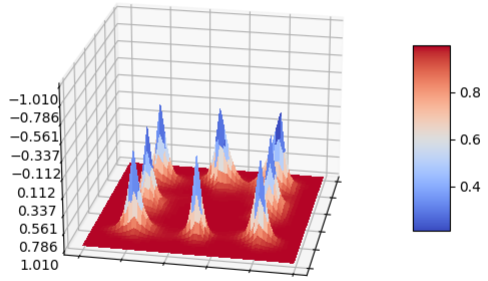
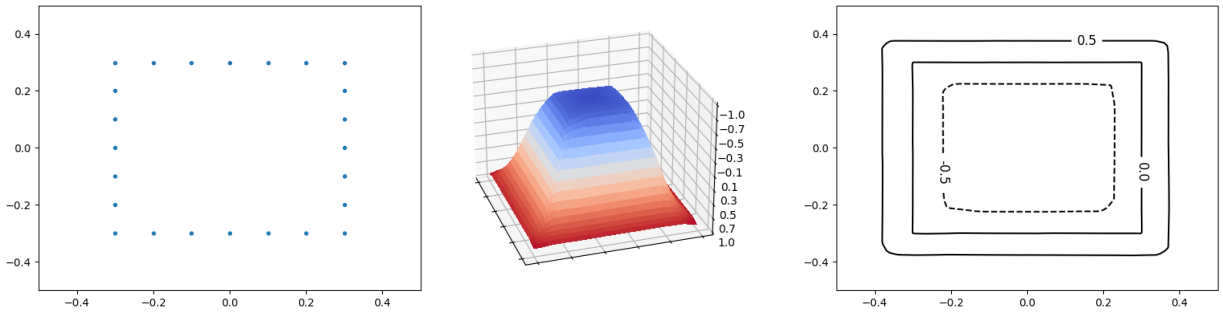


Figure 5.5: Alternative local minimum for the PHASE-loss

This behavior no longer happens by using the geometrical initialization described before. In this case, the network always converges against the signed distance function of the point cloud. Next, we consider a denser point cloud sampled from the square to see if the results improve.

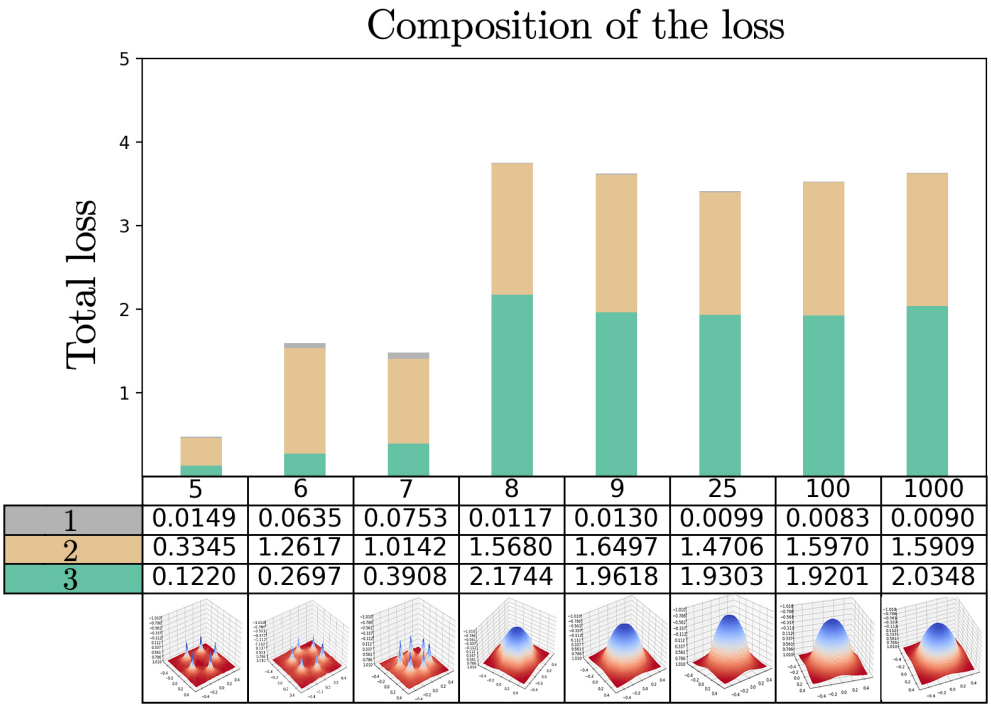


(a) Original point cloud (b) 3D graph (c) Contour plot

Figure 5.6: Reconstructing a square from 24 points

We can see a significant improvement in the contour plot, and the square reconstruction is nearly perfect. Something else that we see especially in the contour lines of $\pm\frac{1}{2}$ is that the corners of the square are slightly rounded. This is because our solution is smooth, but for a perfect square, we would get that the occupancy function can not be differentiable around the medial axis, i.e., the lines that go from the center through the corners. So that means this is the best we can expect.

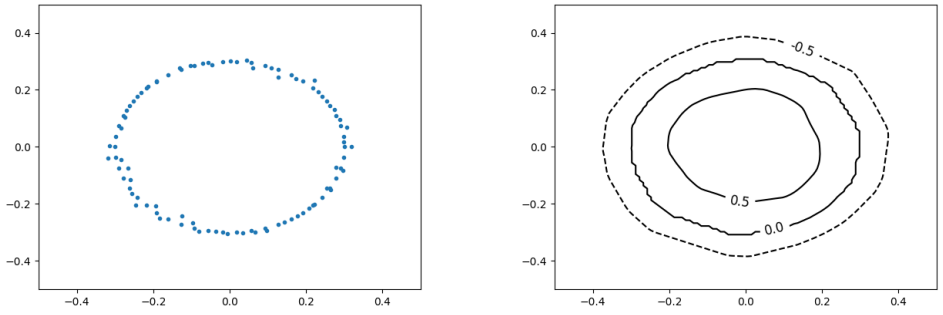
So we have seen that as we increase the number of sampled points, the Modica-Mortola profile is preferred, and the reconstruction improves. To investigate this further, we take n points sampled uniformly from a circle and see how many points it takes on average to get to the Modica-Mortola solution and also how much the different parts of the loss functional contribute. In addition, we explicitly compute the contribution of the three loss terms of the PHASE loss while neglecting the Eikonal term (i.e., setting $\mu = 0$).



	5	6	7	8	9	25	100	1000
1	0.0149	0.0635	0.0753	0.0117	0.0130	0.0099	0.0083	0.0090
2	0.3345	1.2617	1.0142	1.5680	1.6497	1.4706	1.5970	1.5909
3	0.1220	0.2697	0.3908	2.1744	1.9618	1.9303	1.9201	2.0348

Figure 5.7: Reconstructing a circle with radius 0.3 from n points, with $n \in \{5, 6, 7, 8, 9, 25, 100, 1000\}$. The different parts are 1: $\frac{\lambda\varepsilon^{-\frac{1}{3}}}{|\mathcal{P}|}\sum_{p\in\mathcal{P}}\left|\fint_{B_\delta(p)} u(\xi)\,\mathrm{d}\xi\right|$ (grey), 2: $\int_\Omega \varepsilon\|\nabla u(\xi)\|^2\,\mathrm{d}\xi$ (orange) and 3: $\int_\Omega \frac{1}{\varepsilon}W(u(\xi))\,\mathrm{d}\xi$ (green).

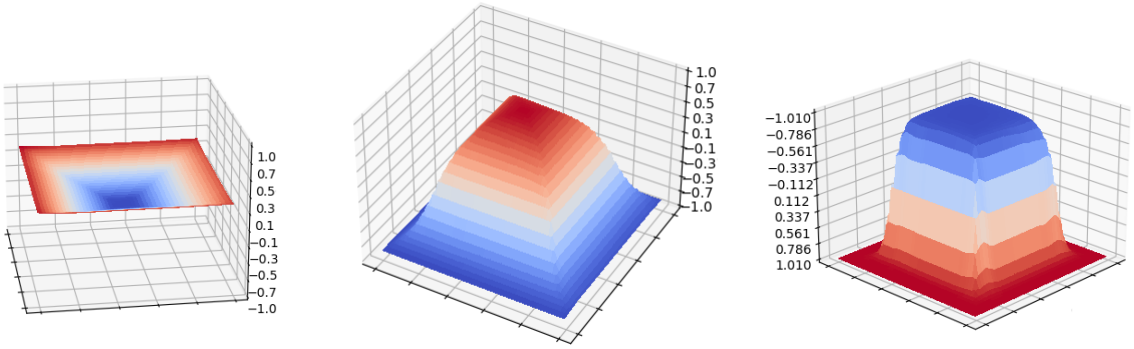
We can see that after 8 points are sampled, the Modica-Mortola profile is achieved. We can also see that the contribution of the gradient term and the double-well potential is relatively equal. In contrast, the contribution of the zero-reconstruction term is relatively small. This is because, for a minimizer, the zero-reconstruction property should always be fulfilled. The Modica-Mortola term, however, does not converge to zero but the scaled perimeter. Also, note that these contributions do not change as we increase the points in the point cloud.
Now we want to see how the algorithm handles noise. Therefore we sample 30 equidistant points from a circle with a radius of 0.3. To every second point, we add noise that is uniformly distributed in $[-0.02, 0.02]^2$.



(a) Original point cloud (b) Contour plot

Figure 5.8: Effect of noise

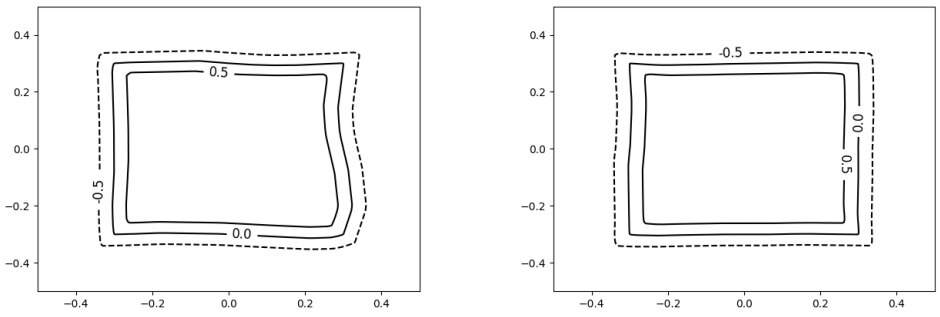
We can see that the algorithm handles noise quite well and can still clearly recognize the circle in the end. Now we want to look at the effect of the different parameters. As we decrease the ε parameter, the contribution of the zero reconstruction loss has to increase.



(a) $\varepsilon = 0.1,\ \lambda = 3.5$ (b) $\varepsilon = 0.01,\ \lambda = 14$ (c) $\varepsilon = 0.001,\ \lambda = 60$

Figure 5.9: Reconstructing a square from 24 points with varying ε parameter

As expected, we can see that the more we decrease the ε parameter, the more the resulting function converges to a proper occupancy function. If ε is too big, the occupancy property nowhere holds since the $\int W(u)$ part in the integral does not contribute much, and the function converges to 0.
If we change the μ parameter on the point cloud with 24 points, we will not see any difference. So to see the Eikonal term's effect, we have to go back to the smaller point cloud.



(a) $\mu = 0$ (b) $\mu = 5$

Figure 5.10: Reconstructing a square from 8 points with varying μ parameter

Here, we can see a slight contour improvement if we choose $\mu > 0$, especially around the corner points. It also prohibits the function from being close to zero in the interior but enforces the phase field property there. Summarizing, enforcing the Eikonal equation makes sense for sparse point clouds but is irrelevant for very dense ones.

5.2.1 3D Examples

Now we can go over to more complicated point clouds. For them, we do need Fourier Features. Otherwise, the results will be too coarse. For our experiments we choose $\sigma = 5$ and $M = 24$,

so we get a total of 48 input features. We also choose $\varepsilon = 10^{-4}$, $\lambda = 50$, $\mu = 0,5$. For the network, we take a three-layered network of each 512 neurons and train it for 50000 epochs. For applying the marching cubes algorithm, we first discretize the function by evaluating it on a grid and then use ParaView software for the visualization, see [Her20]. We use the Fourier-Feature implementation of [Lon21] for our computations.
For testing, we use a point cloud containing 10000 points sampled from the surface of a bunny. The source can be found in the appendix.

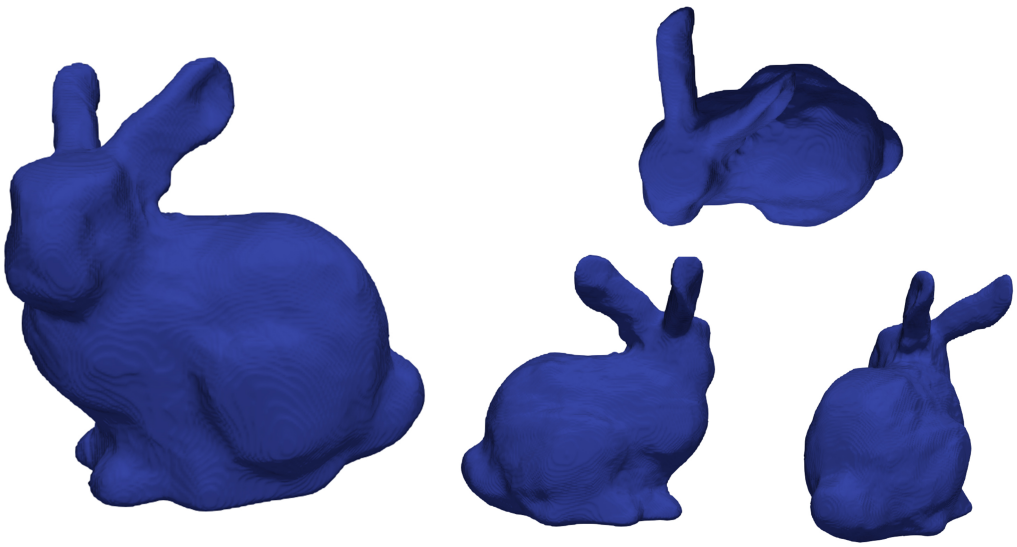
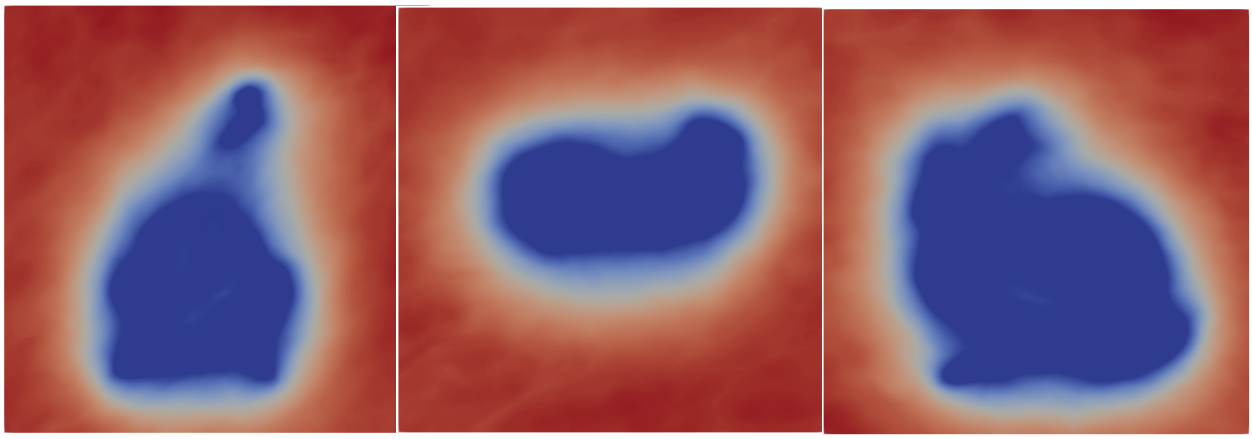


Figure 5.11: Reconstruction of a bunny using the Modica-Mortola-based approach

Next, we consider the cross-sections of this model, where different colors represent different functional evaluations.



(a) Normal: $(1, 0, 0)$ (b) Normal: $(0, 1, 0)$ (c) Normal: $(0, 0, 1)$

Figure 5.12: Cross-sections of the reconstructed bunny using the Modica-Mortola-based approach from different viewpoints

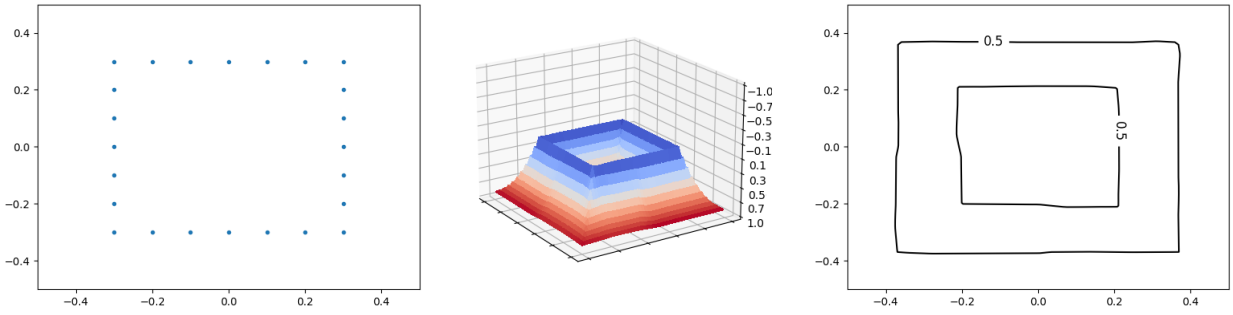
We can see the distinction between inside and outside here. While the function is nearly constant outside and inside the bunny, it makes the jump exactly where the bunny's surface is.

5.3 Ambrosio-Tortorelli-based surface reconstruction

Next, we take a look at some experiments we do now using the Ambrosio-Tortorelli loss, that is

$$\min_{\theta} \int_{\Omega} \frac{1}{4\varepsilon}(v_\theta(\xi)^2 - 1)^2 + \varepsilon \|\nabla v_\theta(\xi)\|^2 \, \mathrm{d}\xi + \frac{\lambda \varepsilon^{-\frac{1}{3}}}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} |v_\theta(p)|. \tag{5.7}$$

We use the same Ω and network parameters for the training as in the 2D Modica-Mortola case. As point clouds, we use the same as before. For these examples we set $\lambda = 2$ and $\varepsilon = 0.01$. We start with the cube point cloud.

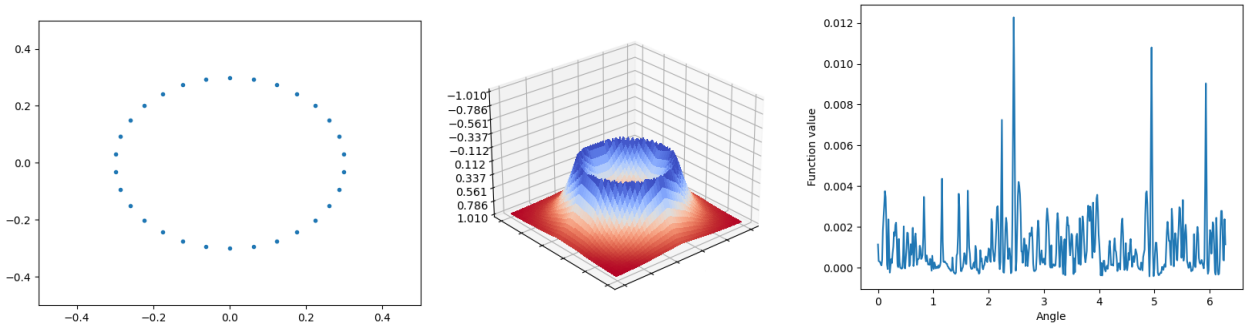


(a) Original point cloud (b) 3D graph (c) Contour plot

Figure 5.13: Reconstructing a square from 24 points

Again, the neural network function is zero along the point cloud, but now the function decreases as one moves more into the interior or exterior of the object. Evaluating the function in some random position, we can no longer tell if we are inside or outside this object. Since we are using marching cubes for extracting the iso-contours, we can no longer extract a 0-level set. However, using the algorithm to extract the iso-contour for $c = 0.5$, we can recognize the shape of the square.

To show how close the function gets to zero, we sample 30 points with uniform distance from a circle centered in the origin and with a radius of 0.3. Then we plot the function values along this circle.



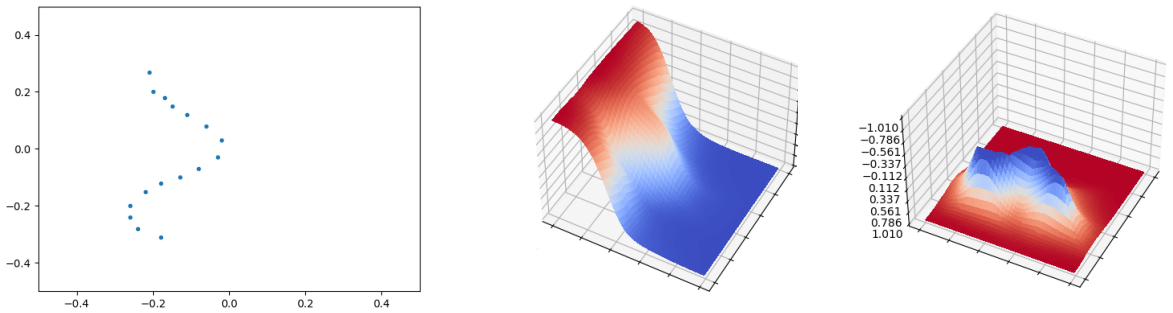
(a) Original point cloud (b) 3D graph (c) Function values along the circle

Figure 5.14: Reconstructing a Circle

We can see that these function values oscillate strongly, but are all very close to 0, except for some outliners, which are not higher than 10^{-2}.

5.3.1 Comparing Ambrosio-Tortorelli with Modica-Mortola

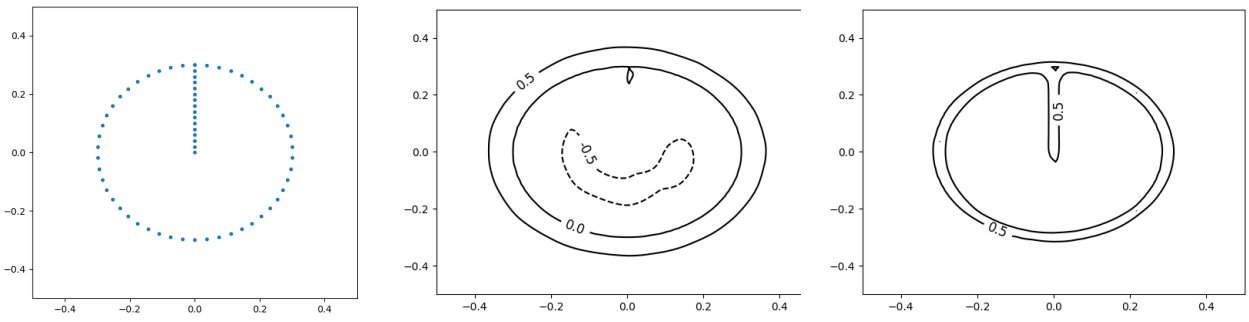
Any reconstruction that works well using the Modica-Mortola approach also gives good results using Ambrosio-Tortorelli, but not the other way around. The big difference between the two loss functionals is that Ambrosio-Tortorelli does not distinguish between interior and exterior. This is especially useful if we consider shapes that are not closed, such as open paths.



(a) Original point cloud (b) Modica-Mortola (c) Ambrosio-Tortorelli

Figure 5.15: Reconstructing an open path

The Modica-Mortola-based approach takes the curve's endpoints and connects them on the shortest path to the domain's boundary. The function changes its sign along the point cloud because of the zero reconstruction loss property. Therefore, the path is seen as a part separating the domain into a left and a right part. In the Ambrosio-Tortorelli case, such a distinction is not made; this can better reconstruct the input curve. Therefore it is better suited for dealing with open objects.
As another example, we consider a circle with a line going from the boundary to the center point.



(a) Original point cloud (b) Modica-Mortola (c) Ambrosio-Tortorelli

Figure 5.16: Reconstructing a sliced Circle

Without the line, there would be a reasonable separation between the interior and exterior. In this case, the Modica-Mortola approach would work. However, a change of sign in the circle's

interior works against the one in the boundary. These problems do not arise for the Ambrosio-Tortorelli-based approach.

5.3.2 3D examples

We want to test our approach on the same bunny as the Modica-Mortola approach. Therefore, we use the same network design as before with $\sigma = 5,\ \lambda = 50,\ \varepsilon = 10^{-4}$.

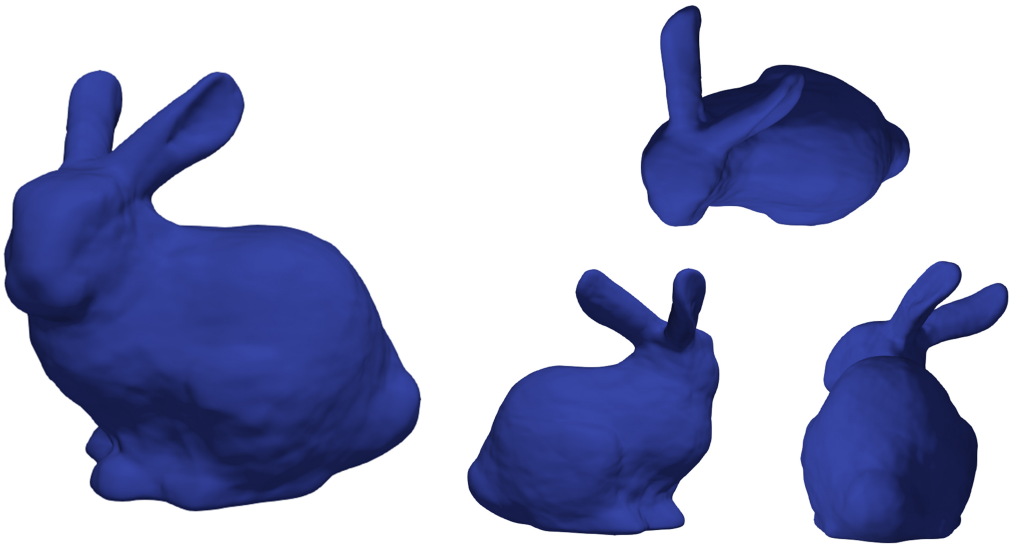


Figure 5.17: Reconstruction of a bunny using the Ambrosio-Tortorelli based loss

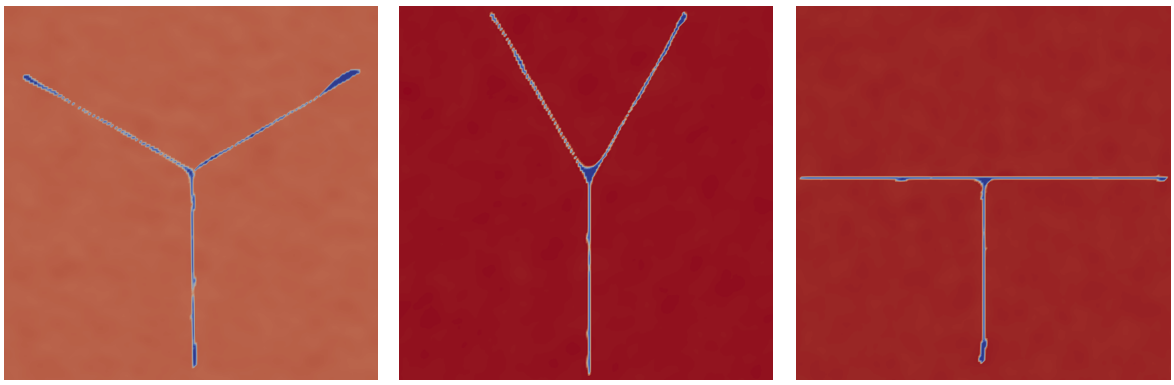
Again, we also consider the different cross-sections of the reconstructed bunny.

(a) Normal: $(1, 0, 0)$ (b) Normal: $(0, 1, 0)$ (c) Normal: $(0, 0, 1)$

Figure 5.18: Cross-sections of the reconstructed bunny using Ambrosio-Tortorelli-based loss from different viewpoints

We can see that the function drops to zero as it reaches the bunny but then goes back to one in the interior.

This enables us to reconstruct open 2D surfaces. For this, we consider the following example. First, we take three closed semicircles and glue them together along their long linear piece at different angles.



(a) $\frac{2\pi}{3}$ (b) $\frac{5}{6}\pi$ (c) $\frac{\pi}{2}$

Figure 5.19: Reconstructing slices at different angles

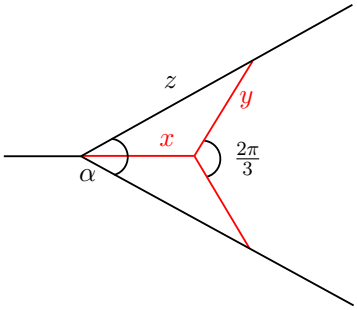
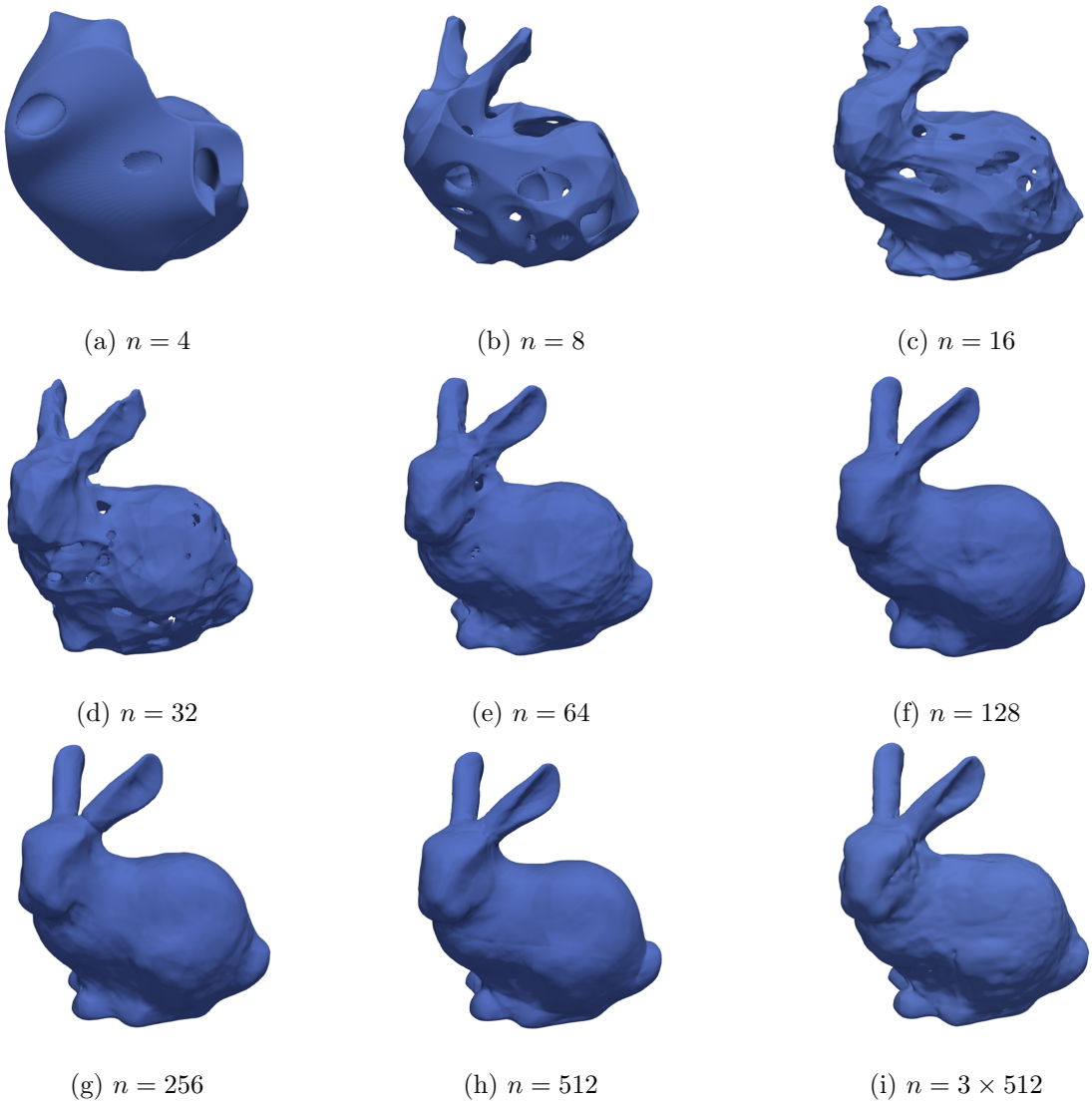


Figure 5.20: Perimeter for slices

As ε tends to zero, the Ambrosio-Tortorelli-based loss does not converge against a minimal perimeter solution, but for a fixed and small ε, we can show that it behaves similarly. We can see this in the profiles of the reconstructed slices, where angles below $\frac{2\pi}{3}$ are being flattened out. This is because, in these cases, we can enter a cross inside with all angles being $\frac{2\pi}{3}$. Then the perimeter is minimized by this cross and not the original shape, i.e., referring to figure (5.20), one can show that $x + 2y < 2z$. The red construction gets filled in because of a limited interface width.

5.3.3 Storage efficiency

Next, we want to find out how big a neural network has to be to give a sufficient reconstruction. To this end, we design a network with 24 Fourier-Features with $\sigma = 4$ and a single hidden layer with n neurons. This gives a total of $48n + n + n + 1 = 50n + 1$ learnable parameters in the network. We choose the other parameters as before.
After training the network with the Ambrosio-Tortorelli-based loss, we plot the resulting isosurfaces. In (5.21i), we also show the results using three hidden layers with 512 neurons each.



(a) $n = 4$ (b) $n = 8$ (c) $n = 16$

(d) $n = 32$ (e) $n = 64$ (f) $n = 128$

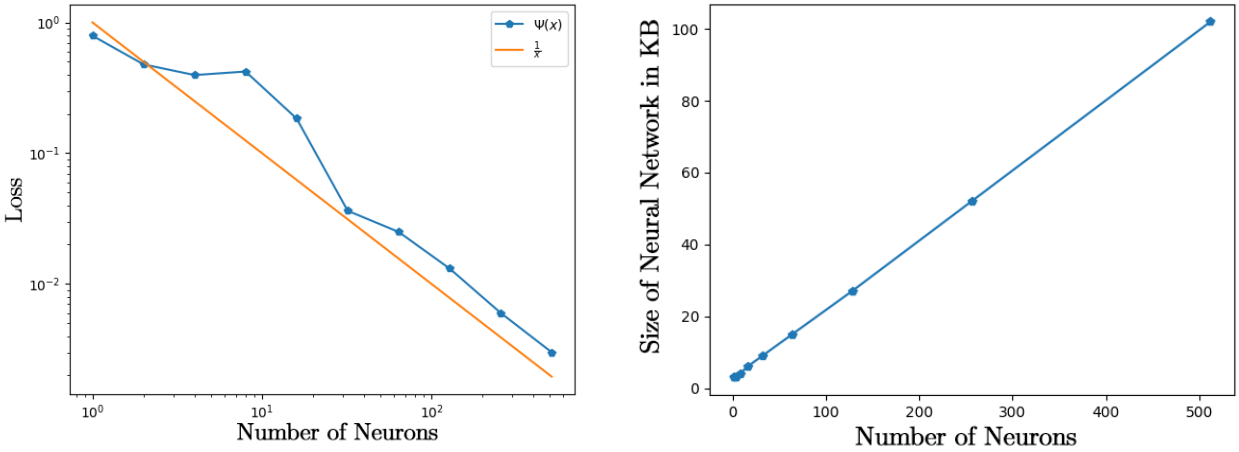
(g) $n = 256$ (h) $n = 512$ (i) $n = 3 \times 512$

Figure 5.21: Reconstructing a bunny with different network sizes

One can see that the reconstruction becomes more accurate with increasing network size. However, already with few neurons, the network represents the coarse structure of the shape. It starts to learn details only if enough parameters are available. Small networks are prone to forming holes in the surface, which become smaller as the network grows. To quantify the results, we define a loss

$$\Psi(v) := \sum_{p \in \mathcal{P}} |v(p)|. \tag{5.8}$$

This loss should converge to zero as the network's size increases. In addition, we also measure the memory sizes of the individual networks and plot them against the number of neurons used.



(a) Accuracy of Neural Networks

(b) Size of Neural Networks

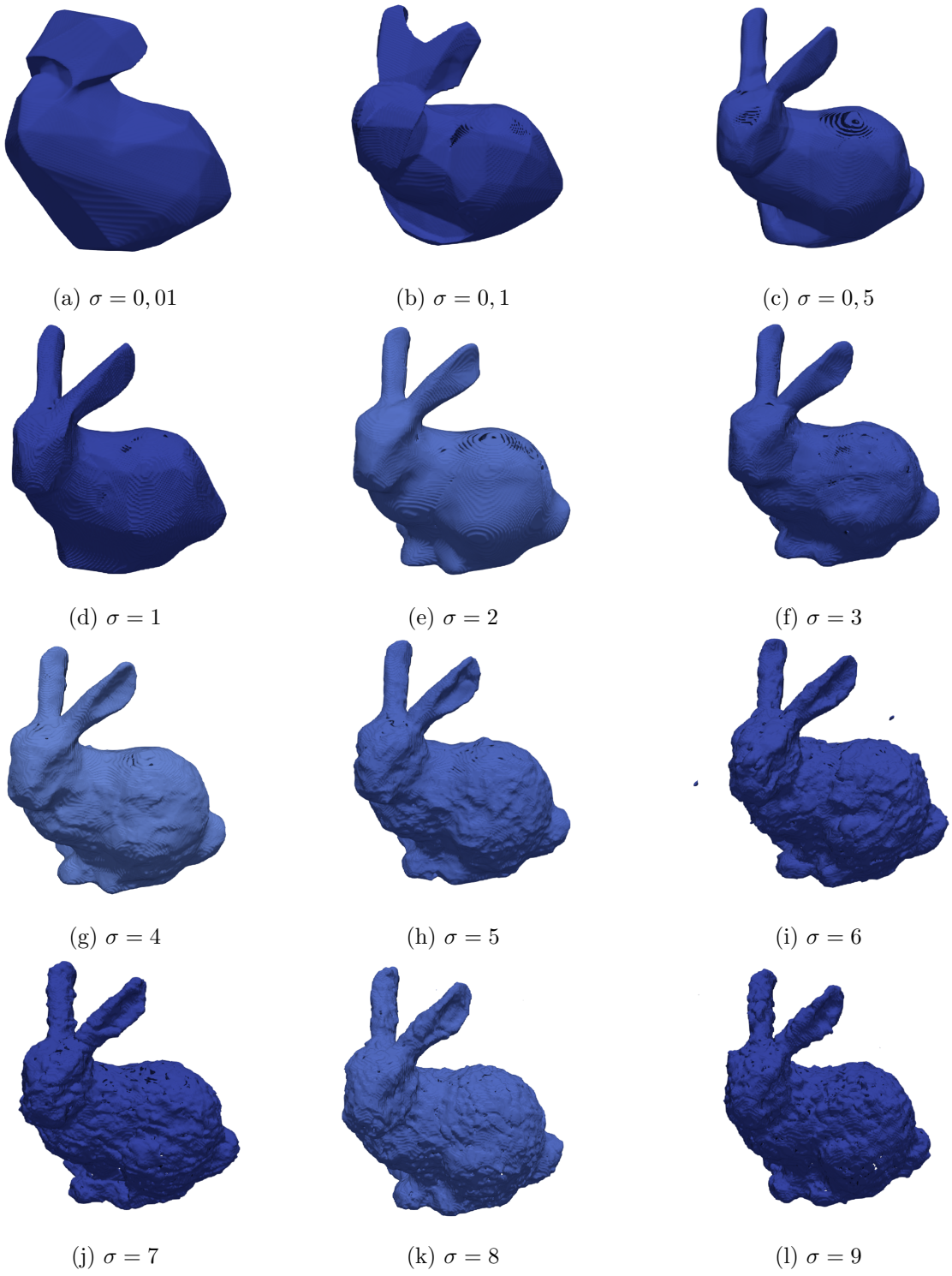
Figure 5.22: Reconstructing a square from 8 points with varying μ parameter

Using a log-log plot, we can see a first-order dependency between the loss Ψ and the number of neurons n. At the same time, the size of the neural network grows linear in its number of neurons because it has only one layer.

The fact that the accuracy of the network increases in the same order that the storage of the network increases makes them an appropriate form of representation for 3D data. It also means that neural networks can be used to compress data by decreasing the network size and retraining it. Thereby, we reduce the required storage capacity but still preserve the rough structures of the object.
In general, this justifies why implicit neural networks can be used for arbitrary data compression if we have a suitable loss-functional. There are several applications for this, for example, in image [SPY+21, DGA+21], audio [ZLO+21], or film compression [DMC+21].

5.3.4 Fourier Features

Now we want to examine the effect of the Fourier features, more precisely, the standard deviation σ. Recall that the Fourier Features are an embedding $\gamma(x) = \begin{pmatrix}\cos(2\pi Bx) & \sin(2\pi Bx)\end{pmatrix}^T$, where the entries of B are samples from a normal distribution $\mathcal{N}(0, \sigma^2)$. Therefore, we take the same network as before and test different parameter values. The parameter σ specifies which frequencies the network should learn.



(a) $\sigma = 0,01$ (b) $\sigma = 0,1$ (c) $\sigma = 0,5$

(d) $\sigma = 1$ (e) $\sigma = 2$ (f) $\sigma = 3$

(g) $\sigma = 4$ (h) $\sigma = 5$ (i) $\sigma = 6$

(j) $\sigma = 7$ (k) $\sigma = 8$ (l) $\sigma = 9$

Figure 5.23: Reconstructing a bunny using different Fourier Features

We see that with a small σ value, the network only learns low-frequency parts of the object and, therefore, only reproduces the coarse structure. However, we can always see an almost watertight surface. If σ is too high, we will see very high-frequency noise along the surface. In general, the correct value depends on the underlying problem. If we want to reconstruct very smooth objects, a relatively low value makes sense, while we prefer a higher value for a more organic object with more texture.

5.3.5 Different 3D Objects

Finally, we want to test our Ambrosio-Tortorelli-based loss functional on various 3D models. The sources for the input point clouds can be found in the appendix. For the experiments, we use a network of three layers with 512 neurons each, 8 Fourier features with $\sigma = 5$, $\lambda = 40$ and $\varepsilon = 10^{-4}$. We start with a few closed objects, beginning with the Armadillo.

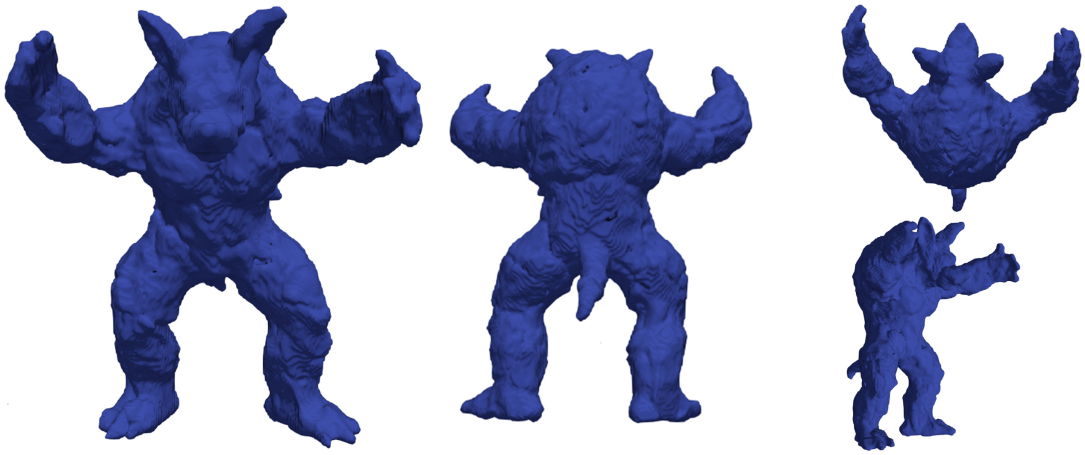


Figure 5.24: Reconstruction of Armadillo from 43243 points

Details such as the fingers or toes can be seen, as can the texture of the back armor. Only in a few places, like the right eye, is the surface a bit imprecise.
Second, we consider a scan of the bust of the Greek orator Demosthenes from the Louvre.

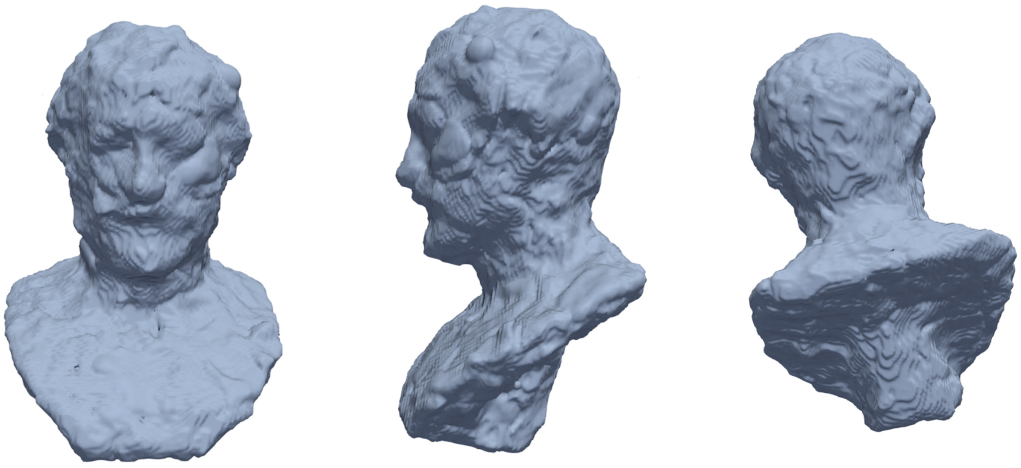


Figure 5.25: Reconstruction of Demosthenes statue from 120309 points

Textures such as the beard or the eyebrows can be easily recognized, but the eyes can no longer be seen here.
The third object is a resting falcon.

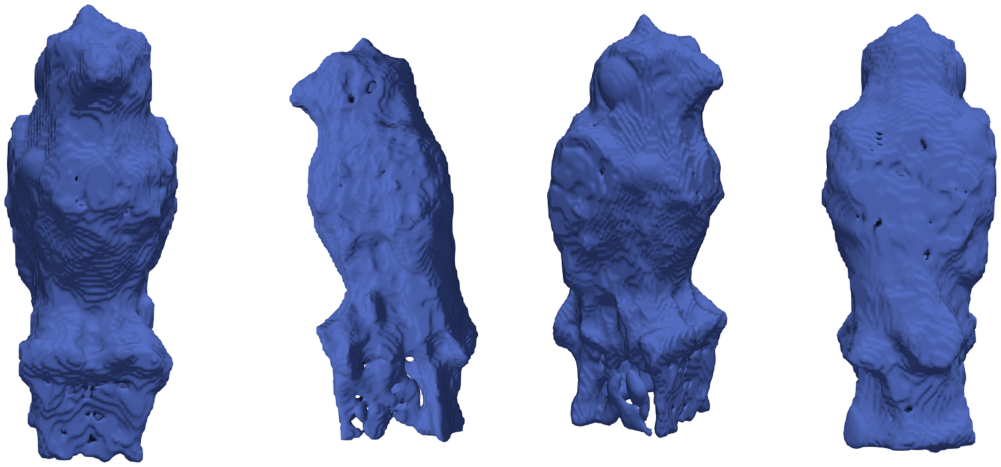


Figure 5.26: Reconstruction of a falcon from 25791 points

In this case, the original point cloud is significantly smaller than the other examples. Again, we can see this on the surface because details on the object are harder to discern.

The following two examples again come from the animal world: a koala and a scorpion. The two point clouds are roughly the same size.

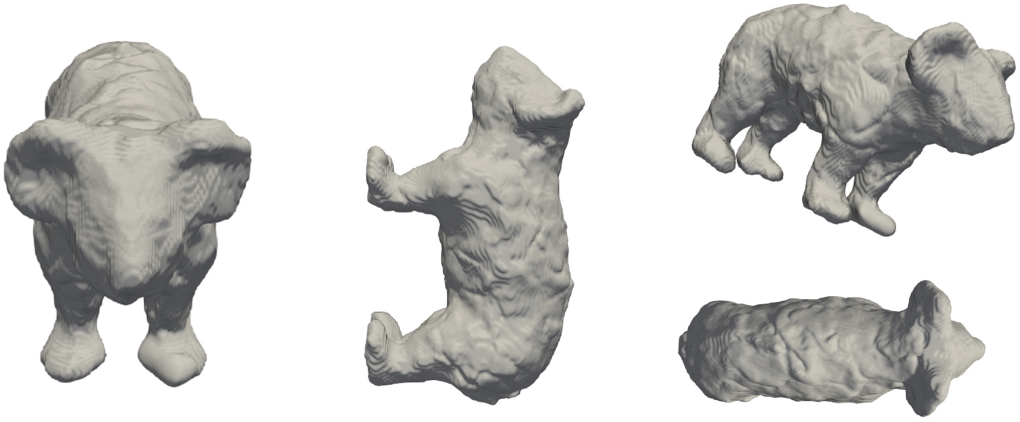


Figure 5.27: Reconstruction of a koala from 49999 points

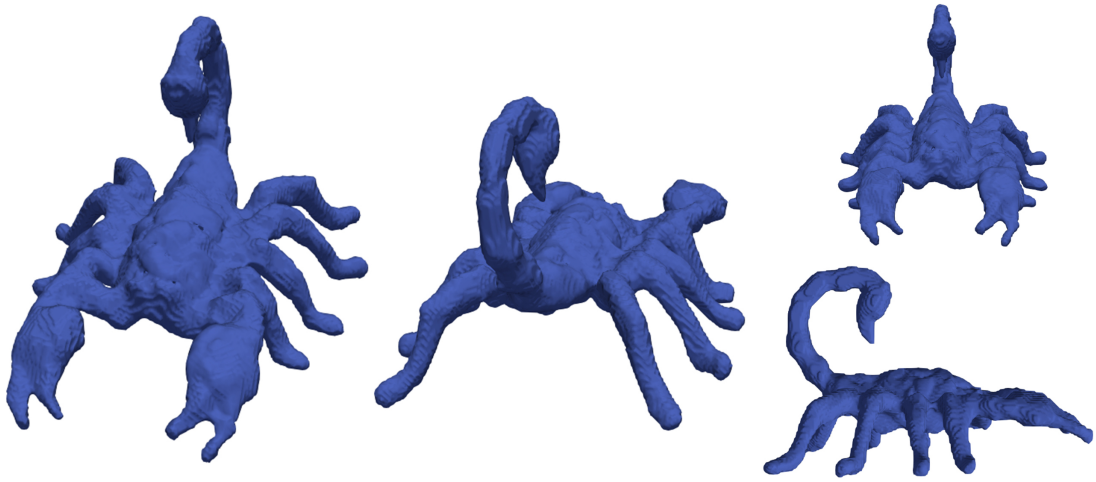


Figure 5.28: Reconstruction of a scorpion from 49997 points

Both reconstructions are almost perfect. The rough texture of the fur can be seen on the koala, and the legs and stinger on the scorpion are clearly visible.
The next object is a scan of a pizza.

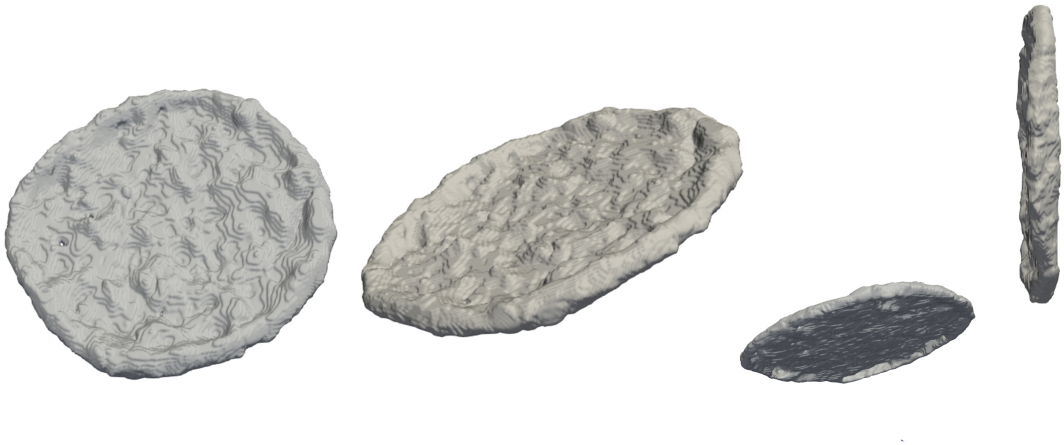


Figure 5.29: Reconstruction of a pizza from 24923 points

We can identify the shape of the pizza but not the toppings.
Next, we consider the relief of a mountain. This is a two-dimensional manifold, another example of a non-closed object that would not work with the Modica-Mortola approach. In that case, the ends of the mountain would be directly connected to the boundary of the Ω domain.

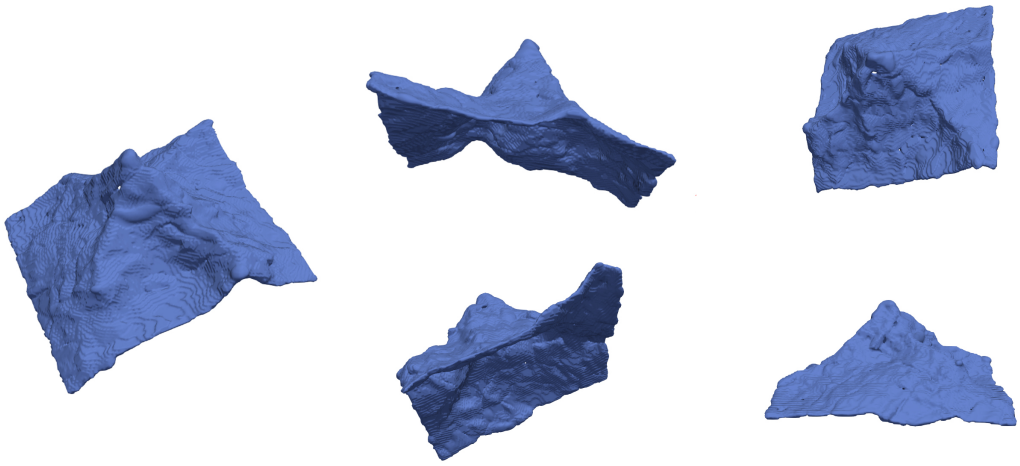


Figure 5.30: Reconstruction of a mountain from 25600 points

The following example is again unsuitable for a Modica-Mortola-based approach. We consider a scan of the Nefertiti statue and remove all points from a cylinder with a diameter of 0.07.

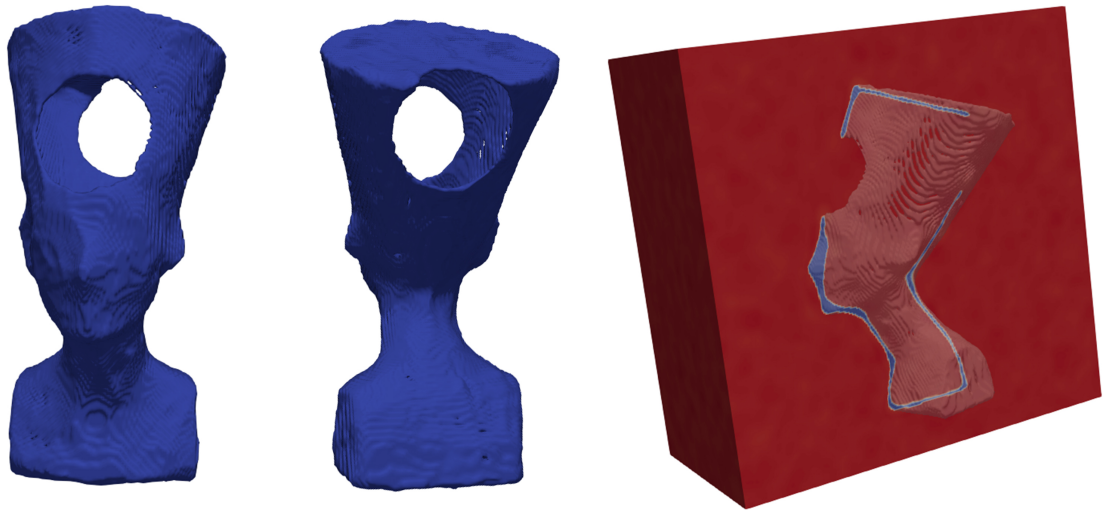


Figure 5.31: Reconstruction of a Nefertiti statue from 1009118 points

Again, the properties of Ambrosio-Tortorelli allow an ideal reconstruction.

Now we look at a point cloud from a horse, where, unlike the other examples, the points are not evenly distributed over the surface. Instead, the points are dense in places with a high level of detail, such as the head, while the torso has few points.

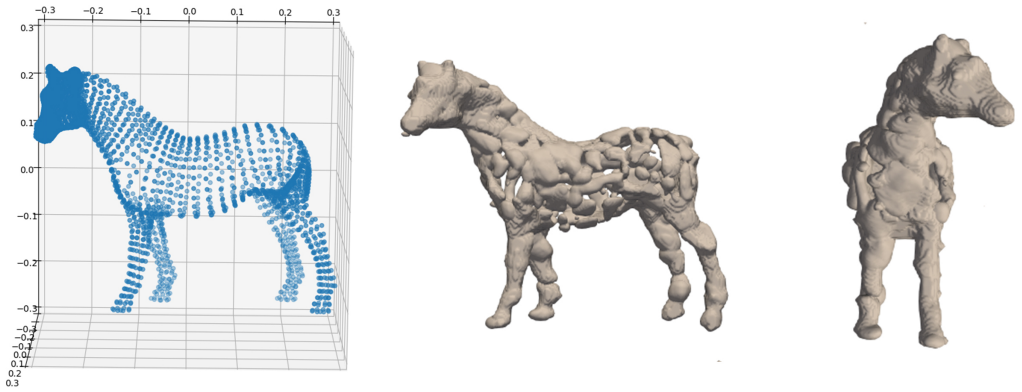


Figure 5.32: Reconstruction of a horse from 2400 points

The procedure fails to give a good reconstruction here because the parameter ε, which is responsible for the width of the phase transition, is the same everywhere. However, it would have to be significantly larger in the areas with only a few points than in the areas with many details.

Finally, we test our method on a few smoother objects. To this end, we choose a lower value for the Fourier feature parameter, namely $\sigma = 3$.

Figure 5.33: Reconstruction of a skull, octopus, and strawberry

One can see that the reconstructed surfaces are much more even than before, and textures are no longer recognizable.

Chapter 6

Shape Space learning

Finally, we want to address the problem of shape space learning. To our knowledge, there has been no attempt to learn a shape space from pure point clouds, which does not include retraining the model for every shape, such as (1.25). Recall that our goal is to train a single network to learn the phase field representations of a whole manifold of shapes. By (1.24), we define a suitable loss based on the Ambrosio-Tortorelli loss for surface reconstruction. To this end, we enter an additional feature vector into the network. To test this, we start by learning a space where we clearly understand what the latent space looks like.

6.1 Ellipsoid Shape Space

One of the easiest manifolds of 3D objects is the space of **ellipsoids**. An ellipsoid is a sphere that is stretched along the three coordinate axes and defined implicitly by the following equation

$$\mathcal{E}(a,b,c) := \left\{ (x,y,z)^T \in \Omega \mid \left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 + \left(\frac{z}{c}\right)^2 = 1 \right\}. \tag{6.1}$$

The parameters a, b, and c measure how much the ellipsoid extends into the direction of the three coordinate axes. Therefore, the space of ellipsoids is three-dimensional, where the feature vector is given by (a, b, c). From this representation, it is also possible to, for example, directly compute the signed distance to the object, see [Ebe13]. However, we want to use our Ambrosio-Tortorelli ansatz for this.

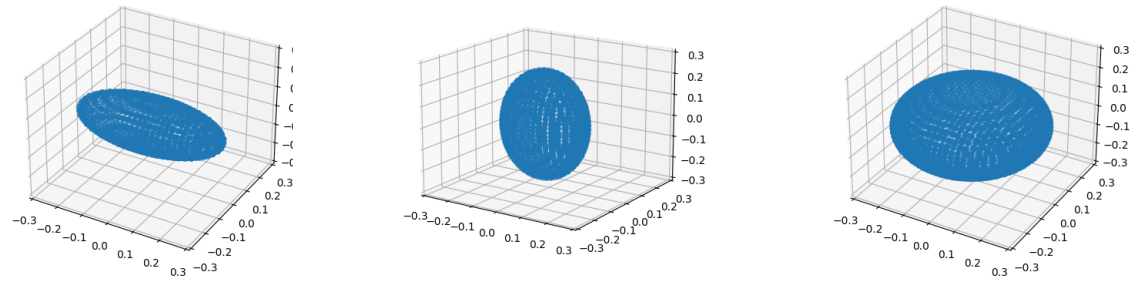
We start by creating ellipsoids by randomly sampling values for a, b, and c uniformly between 0.1 and 0.3. Then we use the Marching Cubes algorithm to transform the implicit shape into a mesh and sample points from its surface. We then store the point cloud $\mathcal{P}_i$ and the three parameters $a_i,\ b_i,\ c_i$ into a dataset.

By (1.24), we have

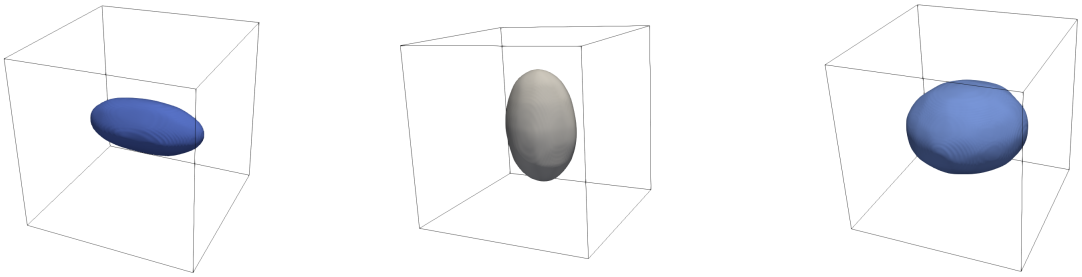
$$\min_\theta \sum_i \left(\int_\Omega \frac{1}{4\varepsilon}(v_\theta(\xi, a_i, b_i, c_i) - 1)^2 + \varepsilon \|\nabla v_\theta(\xi, a_i, b_i, c_i)\|^2 \, \mathrm{d}\xi + \frac{\lambda \varepsilon^{-\frac{1}{3}}}{|\mathcal{P}_i|} \sum_{p \in \mathcal{P}_i} |v_\theta(p, a_i, b_i, c_i)| \right). \tag{6.2}$$

For the parameters, we choose $\varepsilon = 10^{-4}$ and $\lambda = 40$. We create a dataset of 50 shapes, but since the program can not handle all point clouds at once, we only minimize over a batch of

16 shapes in every training iteration, drawn randomly from the dataset. For the shape space learning network, we use an MLP with seven layers of 520 neurons each, 8 Fourier features with $\sigma = 3$, and a skipping layer into the fourth layer. Before entering a point into the network, we concatenate it with the saved feature vector (a_i, b_i, c_i). Therefore the input layer consists of six neurons. We train the network for 70000 with an initial learning rate of 0.01 that we decrease if the loss does not improve after 1500 steps.
We end up with the following results.



(a) GT $(0.29, 0.12, 0.12)$ (b) GT $(0.17, 0.12, 0.26)$ (c) GT $(0.28, 0.28, 0.19)$



(d) Result $(0.29, 0.12, 0.12)$ (e) Result $(0.17, 0.12, 0.26)$ (f) Result $(0.28, 0.28, 0.19)$

Figure 6.1: Learning the space of ellipsoids. Above are the ground truth point clouds with parameters (a, b, c), and below are the results of the shape space learning network.

We can see that the reconstructed shapes perfectly match the original point clouds; therefore, the network has successfully learned shape space.
The ellipsoid shape space is with only three dimensions relatively small. We also want to be able to consider higher dimensional spaces. To this end, we consider a new shape space.

6.2 Metaball Shape Space

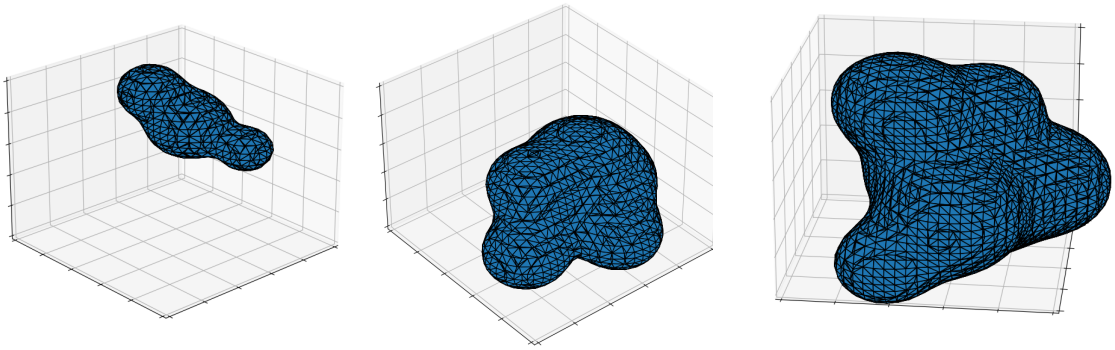
Now we consider another class of objects called Metaballs, see [Ken14]. Metaballs are geometric objects that arise from combining spheres and smoothly blending their boundaries. Each sphere surrounds a density field, where the density decreases as the further one moves away from its center. A ball is given as the level set of the density function. Therefore they are only given implicitly. One enormous strength is that they can be easily combined by adding different density functions. Because these objects are straightforward to handle, they were prevalent in the 90s for creating audio-visual effects in the demoscene.
Formally a Metaball is defined as

$$\mathcal{MB}(s, m, r, g, n) := \left\{ x \in \Omega \mid \sum_{k=1}^{n} \frac{s_k}{\|m_k - x\|^g} = r \right\}. \tag{6.3}$$

Here n is the total number of spheres within one metaball. For every sphere, m_k denotes its origin and s_k its scale. The value g measures the connectivity of two different spheres. The isosurfaces are denoted by r. One can even compute the normal vector of the surface as

$$n_{\mathcal{MB}}(x) = -g \sum_{k=1}^{n} s_k \frac{m_k - x}{\|m_k - x\|^{g+2}}. \tag{6.4}$$

Note that this works in arbitrary dimensions, but we only consider the three-dimensional case. Furthermore, we fix $g = 2$ and $r = 0.8$. One advantage of this approach is that by choosing n, we can define the dimension of the submanifold and adjust the size to our underlying problem. Since any point consists of an origin and a radius, the dimension of the space is given by $4n$.
To create a data set of point clouds, we can now randomly draw s_k and m_k to generate smooth geometric objects. To extract the isosurface, we use the Marching Cubes algorithm again.



(a) $n = 3$ (b) $n = 5$ (c) $n = 8$

Figure 6.2: Examples for metaballs.

We can get the point cloud by sampling points from the surface. Now we could learn the shape space as before using the actual parameters m_k and s_k, but what happens if the real feature vectors are not known priorly?

6.3 Autoencoder

There are many ways to extract a feature vector from an underlying point cloud. One way that has also been used in [MON+19] and [PNM+20] also includes deep neural networks by using **autoencoders**.

Autoencoder networks compress their input data to a low dimensional space and then try to reproduce the original input from this reduced information. Therefore, these networks consist of two different parts. The first one is the encoder E_ψ, a fully connected network, where the number of output neurons is significantly lower than the number of inputs. It maps the input to the feature space, whose dimension we denote by m. The second part is called a decoder D_ϕ, which is a fully connected network whose number of outputs should equal the number of inputs from the encoder. The decoder should reproduce the original point cloud from a feature vector. We train the composition $D_\phi \circ E_\psi$ of these two networks so that the resulting network approximates the identity map. After training, we can enter the point clouds $\mathcal{P}_i$ into the encoder part to obtain the feature vector.

One can improve this method by choosing E_ψ not just to be a simple, fully connected network but one that fits better with the properties of point clouds. One problem is that since a point cloud is an unordered set of points, ideally, the network's output should not change if the order of the points changes. To this end, we use a PointNet encoder to extract the feature vector, see [QSMG17, QYSG17]. PointNet is a class of neural network functions that take a point cloud $\mathcal{P} = \{p_i\}$ as input and can be written in the form of

$$E_\psi = f_{\psi,1} \circ MAX_i\{h_{\psi,2}(p_i)\}, \tag{6.5}$$

where f and h are multi-layer perceptrons and MAX is the max pooling function, the component-wise maximum of a set of vectors. One can show that a sequence of PointNet functions can approximate any arbitrary function. Therefore, one can think of h as the discretization of the input domain and f as the processing function. The big advantage of this type of network is that since MAX is a symmetric operation, the network is by construction invariant under permutations of the points in the point cloud as opposed to an MLP. Therefore, the network can yield much better results. Another advantage of the network is that it can process point clouds of different sizes.

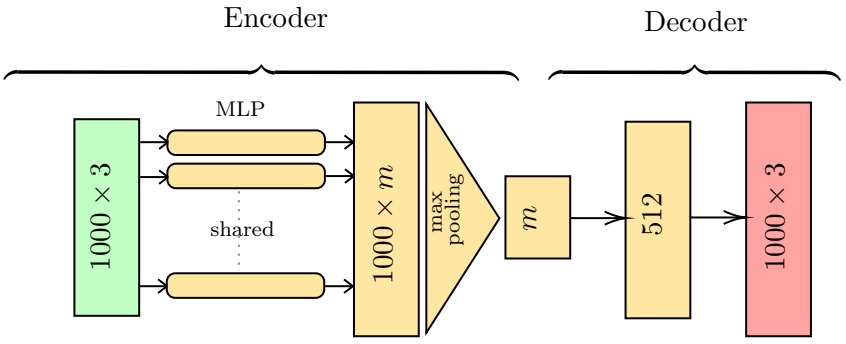


Figure 6.3: PointNet Autoencoder

To train the network, we need a suitable loss functional, that gets minimized for the identity but is again invariant under permutations. We choose the **Chamfer-distance**, that for two point

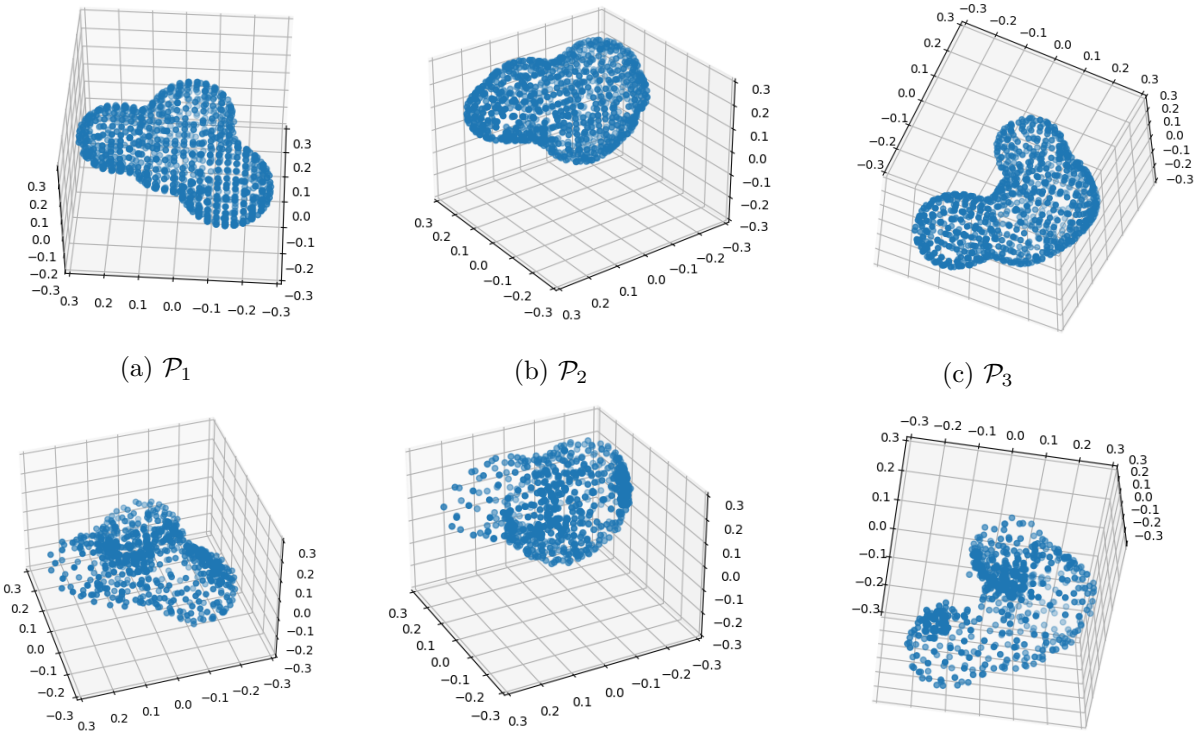
clouds $\mathcal{P}_1$ and $\mathcal{P}_2$ is defined as

$$\mathcal{CD}(\mathcal{P}_1, \mathcal{P}_2) := \frac{1}{|\mathcal{P}_1|} \sum_{x \in \mathcal{P}_1} \min_{y \in \mathcal{P}_2} \|x - y\|_2^2 + \frac{1}{|\mathcal{P}_2|} \sum_{y \in \mathcal{P}_2} \min_{x \in \mathcal{P}_1} \|x - y\|_2^2. \tag{6.6}$$

For every point in the first point cloud, this loss sums up the distance to the closest point in the second point cloud and vice versa. The Chamfer distance is not a metric since it does not fulfill the triangle inequality.
We can then denote the final Auto-Encoder learning problem as

$$\min_{\psi,\phi} \sum_i \mathcal{CD}\big(\mathcal{P}_i,\ (D_\phi \circ E_\psi)(\mathcal{P}_i)\big). \tag{6.7}$$

To test this, we create a dataset of 50 metaballs of different sizes and train the network as shown in (6.4) for 5000 epochs. For the implementation, we use [Suv21] for the Auto-encoder and [RRN+20] for the Chamfer distance.



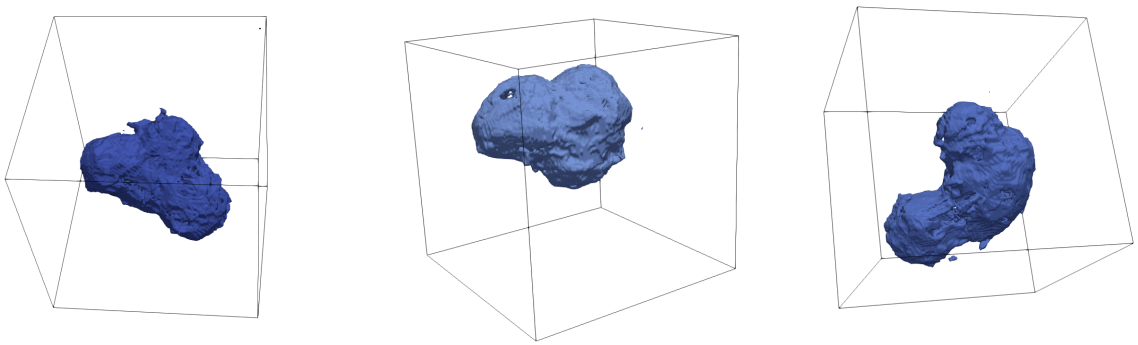
(a) $\mathcal{P}_1$ (b) $\mathcal{P}_2$ (c) $\mathcal{P}_3$

(d) $(D_\phi \circ E_\psi)(\mathcal{P}_1)$ (e) $(D_\phi \circ E_\psi)(\mathcal{P}_2)$ (f) $(D_\phi \circ E_\psi)(\mathcal{P}_3)$

Figure 6.4: Learning the shape space of Metaballs using an autoencoder

6.4 Results

Now one way to handle the shape space learning problem is by separately training an Auto-Encoder as described in the last chapter and using the extracted features as additional inputs for the shape space learning network. However, we observed that this does not yield satisfactory results.



(a) $\{x \mid v_\theta(x, z_1) = 0\}$ (b) $\{x \mid v_\theta(x, z_2) = 0\}$ (c) $\{x \mid v_\theta(x, z_3) = 0\}$

Figure 6.5: Training a shape space learning network separately on the point clouds from figure (6.4)

Depending on the different parameters, they either contain high-frequency noise or are too smooth. This indicates that feature space learned by the Auto-Encoder setup for point clouds does not work well with the Ambrosio-Tortorelli loss. Hence, we want to extract specifically designed features for this task. Therefore, we concatenate the two networks and train both at the same time, using only the Ambrosio-Tortorelli loss.
Before we continue with the results, let us summarize what we have done so far.
At first, we created a dataset of point clouds $\{\mathcal{P}_i\}_{i=0}^N$. Therefore, for every $\mathcal{P}_i$, we drew random parameters of the metaball definition, scaled the object to fit in Ω, and converted the implicit representation into an explicit using marching cubes. Then we sampled 1000 points from the surface and stored them.
Then we considered a PointNet Encoder E_ψ and a network v_θ with Fourier Features as we used for the surface reconstruction problem, which consists of seven fully connected layers and one skipping layer into the fourth one. The shape space learning network v_θ gets as input a point in $x \in \Omega$ and a feature vector as the output from the encoder. Both networks are now trained simultaneously.

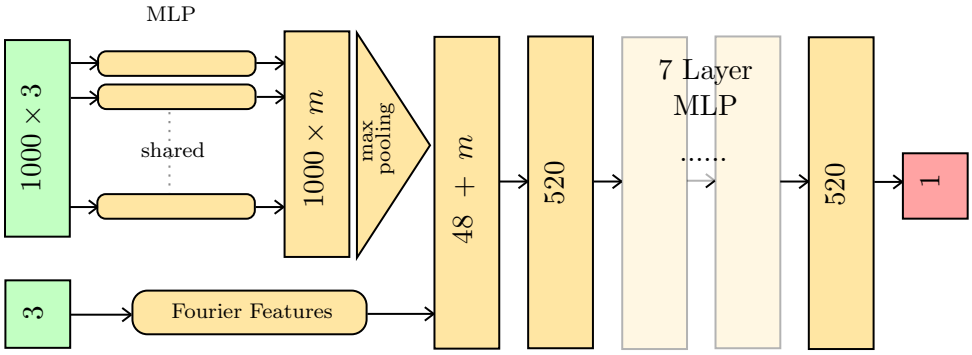
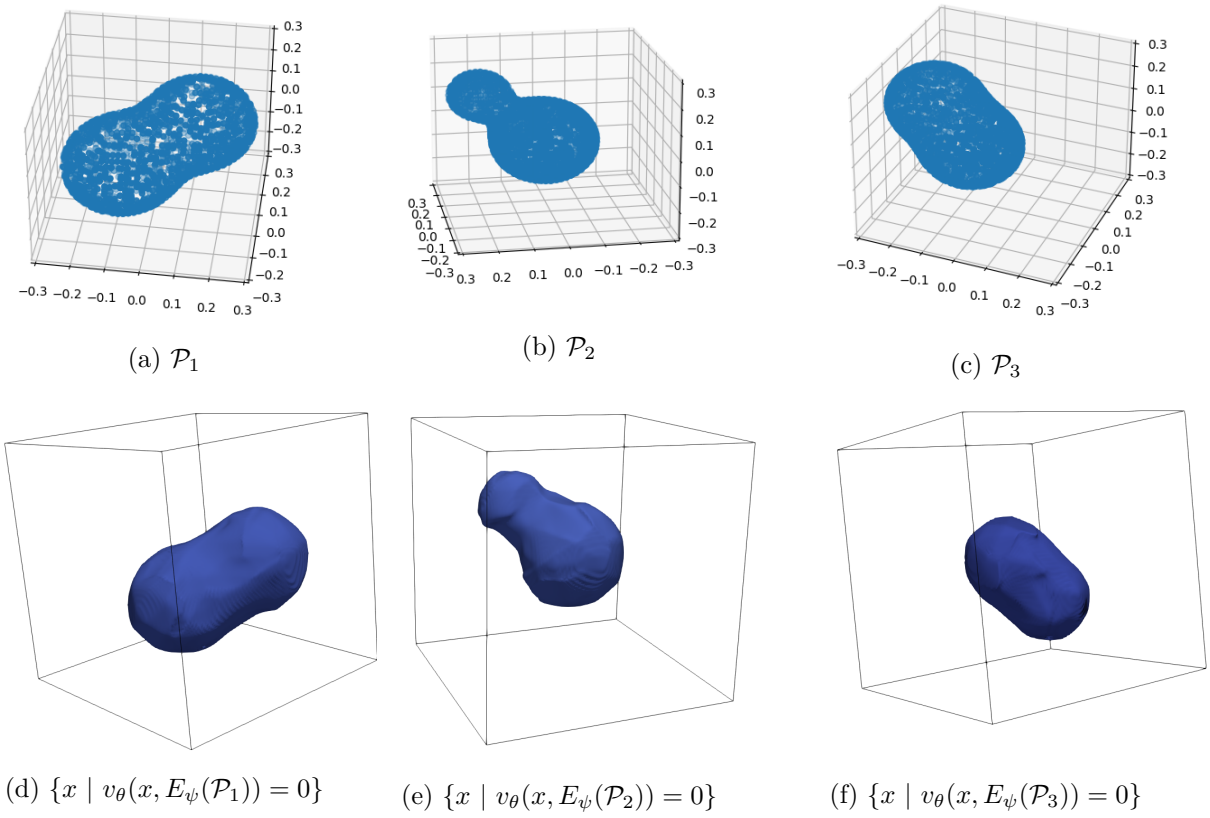


Figure 6.6: Shape space learning network with PoinNet Encoder setup

The output $v_\theta(x, E_\psi(\mathcal{P}_i))$ should be the phase field evaluation of shape $\mathcal{P}_i$ in point x. According to (1.24), we get the following minimization problem

$$\min_{\theta,\psi} \sum_{i=0}^{N} \left(\int_\Omega \frac{1}{4\varepsilon} \left(v_\theta\left(\xi, E_\psi(\mathcal{P}_i)\right) - 1 \right)^2 + \varepsilon \left\| \nabla v_\theta\left(\xi, E_\psi(\mathcal{P}_i)\right) \right\|^2 \mathrm{d}\xi + \frac{\lambda \varepsilon^{-\frac{1}{3}}}{|\mathcal{P}_i|} \sum_{p \in \mathcal{P}_i} \left| v_\theta\left(p, E_\psi(\mathcal{P}_i)\right) \right| \right). \tag{6.8}$$

At first, we test our setup on a dataset of two Metaballs. The resulting feature dimension is $m = 8$.

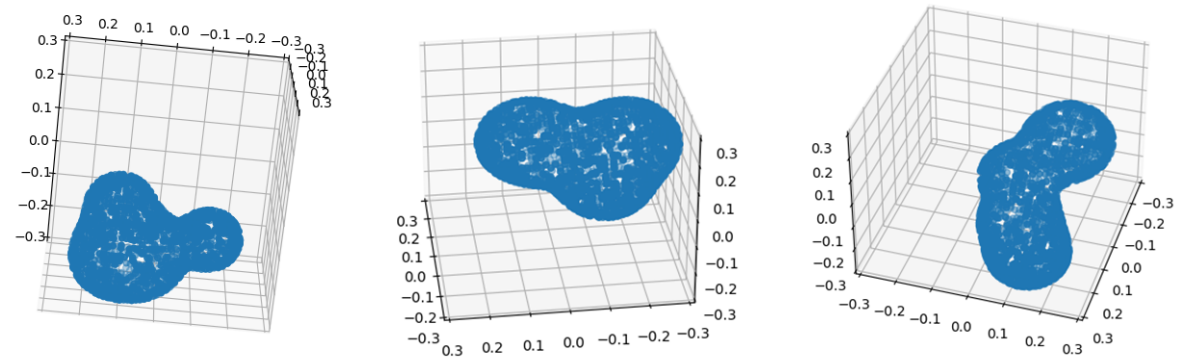


(a) $\mathcal{P}_1$ (b) $\mathcal{P}_2$ (c) $\mathcal{P}_3$

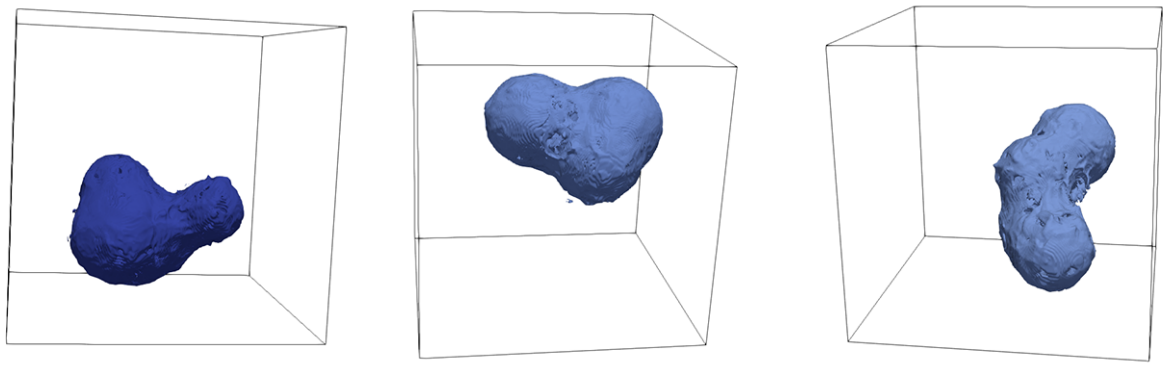
(d) $\{x \mid v_\theta(x, E_\psi(\mathcal{P}_1)) = 0\}$ (e) $\{x \mid v_\theta(x, E_\psi(\mathcal{P}_2)) = 0\}$ (f) $\{x \mid v_\theta(x, E_\psi(\mathcal{P}_3)) = 0\}$

Figure 6.7: Learning the phase field representation an 8-dimensional Metaball shape space

Now we test it for four Metaballs and a feature dimension $m = 16$.



(a) $\mathcal{P}_1$ (b) $\mathcal{P}_2$ (c) $\mathcal{P}_3$



(d) $\{x \mid v_\theta(x, E_\psi(\mathcal{P}_1)) = 0\}$ (e) $\{x \mid v_\theta(x, E_\psi(\mathcal{P}_2)) = 0\}$ (f) $\{x \mid v_\theta(x, E_\psi(\mathcal{P}_3)) = 0\}$

Figure 6.8: Learning the phase field representation a 16-dimensional Metaball shape space

We can see that the reconstructed surfaces are significantly smoother in the smaller of the two shape spaces than in the larger one. Also, the faces of 16-dimensional space sometimes have small holes. Nevertheless, the original shapes can be clearly recognized in both cases.

Chapter 7

Outlook

In this thesis, we have successfully developed a new variational approach for the surface reconstruction problem using phase fields and extended it to the shape space learning problem. In contrast to previous methods, this approach is particularly well suited to reconstruct open surfaces as it no longer distinguishes between interior and exterior points. We have tested this approach and the effects of the individual parameters on various objects and compared it to another method. Moreover, we have found a way to provide phase field queries from various shapes by training two networks simultaneously. For this, the additional network, which is based on a PointNet-autoencoder, learns the latent representation of the input shape. After training, the feature extraction works instantly, and no extra optimization step is needed when we want to access the phase field function of a particular object.

The shape space learning approach could benefit many tasks where point cloud processing and learning are needed, like in autonomous driving, see [CLF+20]. This area uses point cloud reconstruction for a compact representation of the original inputs. One could also use the reconstructed surface for upsampling and downsampling of the point cloud.

One problem that we have seen arises if the latent dimension of objects we want to consider is too high. This could be because we compute thousands of gradients in the direction of a single shape in every training step but only some across the shape space. So if the number of features is very high, this could lead to a somewhat imprecise descent. One way to build on this thesis is to work around this problem by weighting the gradients differently.
A completely different way to improve the results is to improve the Ambrosio-Tortorelli-based functional. For example, similar to the PHASE-loss, one could deduce a transformation from the optimal profile, which transforms the minimizers into a distance function, and then enforce compliance with the Eikonal equation. A problem with this is that the optimal solution is not differentiable along the surface, so one would have to do that in the vicinity of the surface. Another thing one can look at is what happens if the original point cloud consists of more features than the coordinates. For example, one could consider including normal or color information in the loss.

Another point to consider is whether one can show Γ-convergence for the Ambrosio-Tortorelli-

based approach by increasing the density of the point cloud. Depending on ε, one could consider how dense a point cloud has to be for the reconstruction.

Another step to look at is how to proceed with the created implicit surfaces. In [YBHK21] it was shown that it is possible to do geometry processing algorithms like shape smoothing, sharpening, or deformations entirely by using a neural signed distance function. Finally, one could try to transfer these results to the Ambrosio-Tortorelli phase fields.

Index

List of Figures

Bibliography

[ACSTD07] P. Alliez, D. Cohen-Steiner, Y. Tong, and M. Desbrun. Voronoi-based variational reconstructionof unoriented point sets. In *Computer Graphics Forum (Proc. of the Symposium on Geometry Processing)*, 2007.

[Ada20] Nikolas Adaloglou. Intuitive explanation of skip connections in deep learning, 2020. https://theaisummer.com/skip-connections/.

[Agg18] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer International Publishing, September 2018.

[AL19] Matan Atzmon and Yaron Lipman. Sal: Sign agnostic learning of shapes from raw data. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2565–2574, November 2019.

[AL20] Matan Atzmon and Yaron Lipman. Sald: Sign agnostic learning with derivatives. June 2020.

[Amb01] Luigi Ambrosio. Some fine properties of sets of finite perimeter in ahlfors regular metric measure spaces. *Advances in Mathematics*, 159(1):51–67, April 2001.

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)*. ACM Press, 1968.

[Arn20] Anton Arnold. Variationsrechnung. Lecture notes, July 2020.

[AT90] Luigi Ambrosio and Vincenzo Maria Tortorelli. Approximation of functional depending on jumps by elliptic functional via γ-convergence. *Communications on Pure and Applied Mathematics*, 43(8):999–1036, dec 1990.

[BJKK19] Ronen Basri, David Jacobs, Yoni Kasten, and Shira Kritchman. The convergence rate of neural networks for learned functions of different frequencies. *in Advances in Neural Information Processing Systems 32 (NIPS 2019)*, June 2019.

[Bow81] A. Bowyer. Computing dirichlet tessellations. *The Computer Journal*, 24(2):162–166, feb 1981.

[Bra02] Andrea Braides. *Gamma-Convergence for Beginners*. OXFORD UNIV PR, September 2002. https://www.ebook.de/de/product/2768365/andrea_braides_gamma_convergence_for_beginners.html.

[Can99] Emmanuel J. Candès. Harmonic analysis of neural networks. *Applied and Computational Harmonic Analysis*, 6(2):197–218, mar 1999.

[CBC+01] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3d objects with radial basis functions. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01*. ACM Press, 2001.

[CdCG+22] Salvatore Cuomo, Vincenzo Schiano di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific machine learning through physics-informed neural networks: Where we are and what's next. January 2022.

[Cha00] A. Chambolle. Inverse problems in Image processing and Imagesegmentation: some mathematicaland numerical aspects. Lecture notes, September 2000.

[CLF+20] Siheng Chen, Baoan Liu, Chen Feng, Carlos Vallespi-Gonzalez, and Carl Wellington. 3d point cloud processing and learning for autonomous driving. *IEEE Signal Processing Magazine*, 38.1:68–86, March 2020.

[CMPM20] Julian Chibane, Aymen Mir, and Gerard Pons-Moll. Neural unsigned distance fields for implicit function learning. *Advances in Neural Information Processing Systems (NeurIPS) 2020*, 33:21638–21652, October 2020.

[CZ18] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5939–5948, December 2018.

[Del34] Boris N. Delaunay. Sur la sphère vide. *Bulletin of Academy of Sciences of the USSR. 7*, (6):793–800, 1934.

[DG77] E De Giorgi. γ-convergenza e g-convergenza. *Bollettino dell'Unione Matematica Italiana (5)*, 14:213–220, 1977.

[DG78] Ennio De Giorgi. Convergence problems for functionals and operators. *Proc. Int. Meeting on Recent Methods in Nonlinear Analysis*, pages 131–188, 1978.

[DGA+21] Emilien Dupont, Adam Goliński, Milad Alizadeh, Yee Whye Teh, and Arnaud Doucet. Coin: Compression with implicit neural representations. March 2021.

[DMC+21] Dandan Ding, Zhan Ma, Di Chen, Qingshuang Chen, Zoe Liu, and Fengqing Zhu. Advances in video compression system using deep neural network: A review and case studies. *Proceedings of the IEEE*, 109.9:1494–1520, January 2021.

[DNJ20] Thomas Davies, Derek Nowrouzezahrai, and Alec Jacobson. On the effectiveness of weight-encoded neural implicit 3d shapes. September 2020.

[DZW+20] Yueqi Duan, Haidong Zhu, He Wang, Li Yi, Ram Nevatia, and Leonidas J. Guibas. Curriculum deepsdf. *European Conference on Computer Vision*, pages 51–67, March 2020.

[Ebe13] David H. Eberly. Distance from a point to an ellipse, an ellipsoid, or a hyperellipsoid. In *Geometric Tools, LLC*, 2013.

[Fra12] Martina Franken. *Variational Discretization of Higher Order Geometric Gradient Flows Based on Phase Field Models.* PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 2012.

[Giu84] Enrico Giusti. *Minimal surfaces and functions of bounded variation.* Birkhäuser, Boston, 1984.

[GYH+20] Amos Gropp, Lior Yariv, Niv Haim, Matan Atzmon, and Yaron Lipman. Implicit geometric regularization for learning shapes. February 2020.

[Han19] Boris Hanin. Universal function approximation by deep neural nets with bounded width and ReLU activations. *Mathematics*, 7(10):992, October 2019.

[Har96] John C. Hart. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, December 1996.

[Her20] Paulo Herrera. Pyevtk. *GitHub*, 2020. https://github.com/paulo-herrera/PyEVTK.

[HHEM20] Radu Horaud, Miles Hansard, Georgios Evangelidis, and Clement Menier. An overview of depth cameras and range scanners based on time-of-flight technologies. *Machine Vision and Applications, 27(7), 2016*, pages 1005–1020, December 2020.

[HLC+18] Zeng Huang, Tianye Li, Weikai Chen, Yajie Zhao, Jun Xing, Chloe LeGendre, Linjie Luo, Chongyang Ma, and Hao Li. Deep volumetric video from very sparse multi-view performance capture. In *Computer Vision - ECCV 2018*, pages 351–369. Springer International Publishing, 2018.

[HWS+08] Hans-Jürgen Huppertz, Jörg Wellmer, Anke Maren Staack, Dirk-Matthias Altenmüller, Horst Urbach, and Judith Kröll. Voxel-based 3d MRI analysis helps to detect subtle forms of subcortical band heterotopia. *Epilepsia*, 49(5):772–785, may 2008.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, December 2016.

[IRS17] José A. Iglesias, Martin Rumpf, and Otmar Scherzer. Shape-aware matching of implicit surfaces based on thin shell energies. *Foundations of Computational Mathematics*, 18(4):891–927, jun 2017.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. December 2014.

[Ken14] Ben Kenwright. Metaballs & marching cubes. Technical report, 2014. https://xbdev.net/misc_demos/demos/marching_cubes/paper.pdf.

[KH13] Michael Kazhdan and Hugues Hoppe. Screened poisson surface reconstruction. *ACM Transactions on Graphics*, 32(3):1–13, June 2013.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, August 1987.

[Lip21] Yaron Lipman. Phase transitions, distance functions, and implicit neural representations. *ICML 2021*, June 2021.

[Lon21] Joshua M. Long. Random fourier features pytorch. *GitHub*, 2021. GitHub. https://github.com/jmclong/random-fourier-features-pytorch.

[LP14] Gunther Leobacher and Friedrich Pillichshammer. *Introduction to Quasi-Monte Carlo Integration and Applications*. Springer International Publishing, 2014. Chapter 1.

[MK06] Hugues Hoppe Michael Kazhdan, Matthew Bolitho. Poisson surface reconstruction. *Eurographics Symposium on Geometry processing*, pages 61–70, 2006.

[MM77] Luciano Modica and Stefano Mortola. Un esempio di γ^--convergenza. *Boll. Un. Mat. Ital. B*, 5(14):285–299, 1977.

[MON+19] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, December 2019.

[MS89] David Mumford and Jayant Shah. Optimal approximations by piecewise smooth functions and associated variational problems. *Communications on Pure and Applied Mathematics*, 42(5):577–685, jul 1989.

[MST+20] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65.1:99–106, March 2020.

[Mut10] T. Muthukumar. Introduction to Γ-convergence. In *Workshop on Multiscale Analysis and Homogenization, IISc. Bangalore*, Department of Mathematics & Statistics, Indian Institute of Technology-Kanpur, Kanpur, Uttarpradesh, India, 208016, July 2010. http://home.iitk.ac.in/~tmk/courses/minicourse/Gamma/tmkMSAH.pdf.

[MV99] Kurt Meyberger and Peter Vachenauer. *Höhere Mathematik 2*. Springer-Verlag, 3 edition, 1999.

[PFS+19] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 165–174, January 2019.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, December 2019.

[PNM+20] Songyou Peng, Michael Niemeyer, Lars Mescheder, Marc Pollefeys, and Andreas Geiger. Convolutional occupancy networks. *European Conference on Computer Vision*, Springer, Cham,:523–540, March 2020.

[Pou20] Florent Poux. How to represent 3d data?, May 2020. https://towardsdatascience.com/how-to-represent-3d-data-66a0f6376afb Last Access: 24.11.2021.

[QSMG17] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, December 2017.

[QYSG17] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *Advances in neural information processing systems*, 30, June 2017.

[RBA+18] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred A. Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. *ICML 2019*, June 2018.

[Ric73] A. Ricci. A constructive geometry for computer graphics. *The Computer Journal*, 16(2):157–160, February 1973.

[RRN+20] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3d deep learning with pytorch3d. July 2020.

[Run01] C. Runge. Über empirische funktionen und die interpolation zwischen äuidistanten ordinaten. In *Zeitschrift für Mathematik und Physik*, volume 46, pages 224–243, 1901.

[Sig06] Christian Sigg. *Representation and Renderingof Implicit Surfaces*. PhD thesis, ETH Zurich, 2006.

[SJ22] Nicholas Sharp and Alec Jacobson. Spelunking the deep: Guaranteed queries on general neural implicit surfaces via range analysis. *ACM Transactions on Graphics (TOG)*, 41(4):1–16, February 2022.

[SMB+20] Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation

functions. *Advances in Neural Information Processing Systems*, 33:7462–7473, June 2020.

[SPY+21] Yannick Strümpler, Janis Postels, Ren Yang, Luc van Gool, and Federico Tombari. Implicit neural representations for image compression. December 2021.

[Suv21] Dhiraj Suvarna. Pointnet autoencoder. *GitHub*, 2021. https://github.com/dhirajsuvarna/pointnet-autoencoder-pytorch.

[TO02] Greg Turk and James F. O'Brien. Modelling with implicit surfaces that interpolate. In *ACM Transactions on Graphics 21*, volume 4, pages 855–873. ACM Press, 2002.

[TSM+20] Matthew Tancik, Pratul P. Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T. Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *Advances in Neural Information Processing Systems*, 33:7537–7547, June 2020.

[Var67] Sathamangalam Ranga Iyengar Srinivasa Varadhan. On the behavior of the fundamental solution of the heat equation with variable coeffcients. *Communications on pure and applied mathematics*, 20:431–455, 1967.

[Wei17] E. Weinan. A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 5(1):1–11, March 2017.

[Wol95] Franz-Erich Wolter. Cut locus and medial axis in global shapeinterrogation and representation. MIT, Department of Ocean Engineering, Design Laboratory, October 1995.

[YBHK21] Guandao Yang, Serge Belongie, Bharath Hariharan, and Vladlen Koltun. Geometry processing with neural fields. In *Advances in Neural Information Processing Systems*, volume 34, pages 22483–22497, 2021.

[YKM+20] Lior Yariv, Yoni Kasten, Dror Moran, Meirav Galun, Matan Atzmon, Ronen Basri, and Yaron Lipman. Multiview neural surface reconstruction by disentangling geometry and appearance. *Advances in Neural Information Processing Systems*, 33:2492–2502, March 2020.

[ZLO+21] Neil Zeghidour, Alejandro Luebs, Ahmed Omran, Jan Skoglund, and Marco Tagliasacchi. Soundstream: An end-to-end neural audio codec. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 30:495–507, July 2021.

[ZZW19] Deyun Zhong, Ju Zhang, and Liguan Wang. Fast implicit surface reconstruction for the radial basis functions interpolant. *Applied Sciences*, 9(24):5335, December 2019.

Copyright of Models

armadilo, *bunny*	The Stanford 3D Scanning Repository, 1993 http://graphics.stanford.edu/data/3Dscanrep/
demosthenes	ryanfb, 2016 https://sketchfab.com/3d-models/louvre-demosthenes-photoscan-cf5d9dd9bf4a457c95dc76e17848727b
falcon	colinfizgig, 2013 https://www.thingiverse.com/thing:46631
horse, octopus	libigl, 2018 https://github.com/libigl/libigl-tutorial-data
koala	YahooJAPAN, 2013 https://www.thingiverse.com/thing:182225
mountain	lastloginname, 2015 https://www.thingiverse.com/thing:991578
nefertiti	Jan Nikolai Nelles and Nora Al-Badri, 2016 https://nefertitihack.alloversky.com/
pizza	Rigsters, 2019 https://sketchfab.com/3d-models/pizza-40d50989fec1460f8838b608d999ccd0
scorpion	YahooJAPAN, 2013 https://www.thingiverse.com/thing:182363
skull	YahooJAPAN, 2013 https://www.thingiverse.com/thing:182365
strawberry	GSC_ Nakamura, 2013 https://www.thingiverse.com/thing:153548